An Advanced, Integrated Display System for Small, High Speed Marine Craft

by

Christopher M. King

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Computer Science and Engineering

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 23, 1997

Author _____
Department of Electrical Engineering and Computer Science
May 23, 1997

Approved by _____
Seamus T. Tuohy
Technical Supervisor
The Charles Stark Draper Laboratory

Certified by _____
Professor John Leonard
Thesis Supervisor

Accepted by _____
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

An Advanced, Integrated Display System for Small,
High Speed Marine Craft
by
Christopher M. King

Submitted to the
Department of Electrical Engineering and Computer Science

May 23, 1997

in Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Computer Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

An integrated display system for a small, high-speed marine craft has been developed for monitoring and control of all boat systems, including navigation, internal and external sensors, and communications from multiple displays. The system integrates disparate data sources in a dynamic physical environment necessitating novel solutions for user interaction and information display technology. These solutions address particular problems in information display, instrumentation, human factors and visual perception within the constraints of commercial off the shelf (COTS) hardware and software development environments. Issues of extensibility in regards to new boat systems as well as improved user input methods and devices are also resolved.

Thesis Supervisor: John Leonard
Title: Professor, MIT Department of Ocean Engineering
Technical Supervisor: Seamus T. Tuohy, Ph.D.
Title: Technical Supervisor, The Charles Stark Draper Laboratory

Acknowledgments

This thesis was prepared at The Charles Stark Draper Laboratory under Contract DAAD05-96-C-0049, W. Wyman, Draper Program Manager.

I would like to thank Séamus Tuohy for his tremendous support and encouragement throughout the entire project. It's been fun and I've learned a lot that I will carry with me. I couldn't have asked for a better project advisor.

Thanks also to my thesis advisor, John Leonard, whose advice helped me turn all of this into a real thesis. Thanks, especially, for accepting me on a moment's notice.

To Kevin Toomey, Daryl Dietz, and Bill Wyman, it was a pleasure working with you. Thank you for the trust and respect that you gave me as well as the advice and the knowledge.

Thanks to Mario Santarelli for taking me in last summer and providing me the stepping stone to this project and to Bill McKinney, who advised me on that project and always found time to give me a hand.

To Linda Leonard, thank you for all of your help starting well before this project. You've always been there whenever I needed anything. And to John Turkovich, who got everything started for me at Draper three years ago.

Finally, to my parents, who have made all of my dreams possible, I could spend my whole life and not say thank you enough.

Contents

Chapter 1 - Introduction

The Integrated Bridge System (IBS) has been developed for the U.S. Navy Office of Special Technology at the Charles Stark Draper Laboratory (CSDL) in conjunction with several other companies. Specifically, CSDL had sole responsibility for the design and development of the Graphical User Interface (GUI). In order to ensure rapid prototyping, commercial off the shelf (COTS) hardware and software was used whenever possible. The objective of this thesis is the design and implementation of a GUI for IBS which addresses the issues of user interaction, display of diverse information, human factors, and visual perception. In particular, the complications involved in producing such a GUI for use in high-speed marine vehicles are examined. Additionally, issues of extensibility in regards to future systems as well as improved user input methods and devices will be resolved.

Chapter 2 describes the problems that exist with current instrumentation and control systems on several Navy SEAL vessel and outlines the goals of the IBS project. Chapter 3 provides an overview of the client/server architecture for the IBS system and establishes the scope of the GUI within the overall project. Chapter 4 explains the GUI from a programmer's perspective providing an understanding of the underlying framework. This chapter in conjunction with Appendix A provides the information necessary to make extensions, modifications, and additions to the GUI. Chapter 5 describes the GUI from the designer's perspective, explaining the display screens and

functions that were developed using the framework described in Chapter 4. Chapter 6 covers the evaluation and testing procedures that were followed throughout the project and Chapter 7 draws conclusions from the development process. Chapter 8 explores extensions and improvements that could be made to the system.

Chapter 2 - Statement of Problem

The U.S. Navy currently has several maritime vehicles commonly used by Special Operations Forces, the U.S. Navy SEALs, including (but not limited to) the MK V Special Operations Craft (MK V SOC), the High Speed Assault Craft (HSAC), and in development, the Very Slender Vehicle (VSV). These vehicles are used for missions involving insertion and extraction, Coastal Patrol and Interdiction, and Target Interception [1]. MK V SOC and HSAC, craft that are currently in service, are traditional planing hull vehicles [25,26] while the VSV, currently under development, will be a wave-piercing craft [24]. All of the vehicles can reach speeds exceeding 50 knots and can operate in high sea states resulting in a high shock environment on board [25,26], although the wave-piercing ability of the VSV is an attempt to ameliorate this problem [24].

Operation of these vessels involves a multitude of interfaces, functions, and equipment that must be continually monitored or controlled from a console. While each crew member has primary responsibilities relating to the displays located near them, they also have additional secondary function responsibilities under certain conditions. Since all of the boat systems have separate interfaces made up primarily of electro-mechanical (E-M) instruments, space limitations on the bridge made it impossible to provide all of the these displays for every crew member station [1]. Control of some onboard equipment from the crew stations was sacrificed completely due to this limited bridge

setup. A display system was needed to replace the current bridge instrumentation on these vessels and provide monitoring and control capabilities to all crew member stations.

Limitations of E-M displays and instrumentation were first addressed in both military and civilian aircraft cockpits. In addition to occupying precious space within the cockpit, E-M displays are difficult and expensive to maintain [13]. Beginning over fifteen years ago, there has been a decided trend towards electro-optical (E-O) multi-function displays (MFDs), often referred to as a glass cockpit, with virtually all new aircraft incorporating this type of technology [15]. The Boeing 777, for example, was designed to incorporate E-O MFDs [14]. Studies have shown that this type of display technology has the potential of providing a substantial increase in the pilot's efficiency [13].

Refitting a vehicle with such technology, as was required for the SEAL vessels, has been a successful practice with various aircraft. Military craft such as the F-16 A/B, F-16 C/D [18], F/A-18 E/F Hornet [17], and the F-22 [16] all have been retrofitted with such displays. In fact, the F-22 is the first military aircraft to integrate an exclusively *glass cockpit* [16].

This cockpit display technology has also been extended to other, non-aircraft vehicles. Studies have been made to incorporate E-O MFDs into future ground combat vehicles including the Bradley Infantry Fighting Vehicle and the Abrams Main Battle Tank [23]. In fact, several MFDs proposed for the Abrams Tank utilize a bezel-mounted twenty button interface, very similar to the button interface – described in the next chapter – chosen for the SEAL vessels.

Based on the success of integrated E-O MFDs in a wide range of military and civilian vehicles, the Integrated Bridge System (IBS) was developed to integrate all of the boat systems so that each crew member could have access to all pertinent information and controls.

Chapter 3 - Project Overview

3.1 Requirements

The overall system had a number of requirements reflect the desires of the SEAL crews, tempered by the limitations inherent in retrofitting an existing craft.

- The new system, including all displays, computers, and sensors, needed to fit within the space limitations of the current vessels.
- Displays needed to be readable not only in normal operation but also under direct sunlight and at night.
- The system needed to function in a harsh marine environment.
- Interaction with the system needed to be feasible in this environment while minimizing errors in operation and cognition.
- All of the current bridge displays and controls needed to be represented, including Global Positioning System/Navigation System, Compass, Radar, Electrical Status, Fuel Management System, and Engine/Propulsion Data.
- New functionality needed to be added including Communications, Alarm Notification, and Video.

- The interface to these systems needed to allow the crew members to focus quickly and accurately on situation specific information.
- The system needed to be general enough to be ported easily, in terms of software modifications, to any of the three Navy SEAL craft and to be extended with new boat systems as they are developed.

3.2 Hardware Design

3.2.1 Client/Server Model

The Integrated Bridge System (IBS) is arranged in a client/server model (See Figure 2.1). The central server receives and processes sensor inputs and dispatches control commands to on-board equipment. Through the incorporation of new sensors and control pathways, the server is able to interface with all of the current bridge systems as well as several additional ones, including Communications, Alarm Notification, and Video. Multiple clients – one for each crew member station – running distributed copies of a GUI are connected to the server through a Local Area Network. The server and the clients communicate through a predetermined interface providing access for each client to all of the information maintained by the server.

Standard applications generally involve two conceptual levels — the Application Layer, which handles all of the computation and functionality of the application, and the Interface Layer, which handles presentation and interaction with the user [4]. The clients, in essence, function as the Interface Layer of the IBS system, employing the usual method of call-back functions [12] to interface with the Application Layer, the server. The advantage of this architecture is that all of the clients reflect the state of the server

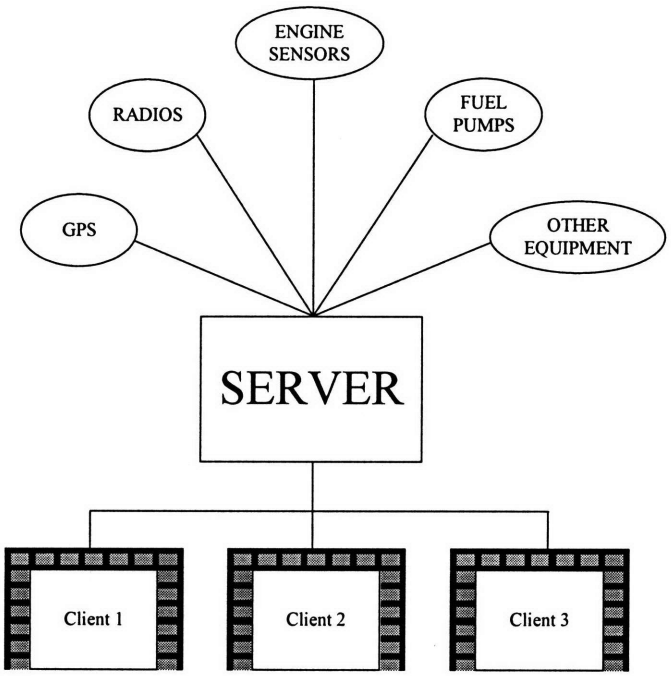

Figure 3.1 - Client/Server Architecture

which, in turn, maintains information about the state of the craft. Any changes in the state of the craft, whether initiated by the GUI or by external forces will not be reflected in the GUI until and unless those changes are registered in the Server's database and communicated to the client. Since each client is querying the same server, the information displayed across the clients is guaranteed to be consistent. Each client,

however, operates independently of the other clients and has its own input devices. This allows the display and control of any information at any location. Issues of concurrent data access and collision of controlling input are addressed by strict serialization of access to the server's data [3].

An interface library was developed to handle all of the communication between the clients and the server (See Appendix B). This library provides functions for acquiring all of the data that can be displayed in the GUI. It also provides the data structures in which the data will be stored and passed.

Communication between the clients and server, imbedded in the interface library, takes place through socket connections established by the clients with the server. The clients can call the interface functions as necessary. In most cases, this means polling the server for data, although functions also exist for requesting a change of state. The server is responsible for verifying that the specified state change is valid and disregarding it if it is not.

The overall project was divided between multiple companies along the natural boundary between client and server. Draper Labs was responsible for the production of the GUI that would operate on the client machines, and the development of this GUI is the main focus of this thesis.

3.2.2 Input Interface

A number of input devices were rejected as unsuitable for this project for a variety of reasons. The standard keyboard/mouse interface was discarded because it is too difficult to manipulate in the rough and dynamic environment. Touch screens were also considered but disregarded. Heat sensing touch screens would be ineffective because the crew members wear gloves and touch screens that register input through physical displacement would be unreliable because high impact shocks to the craft could cause disturbances in the screen resulting in erroneous and dangerous inputs (e.g. for weapons systems). Data gloves were discussed but were rejected by the crew members who did not wish to have any additional constraints put on their hands. Speech recognition was rejected because it is untested in this type of environment and the necessary tests could not be carried out within the time frame and development budget of this project.

Currently, a static button interface consisting of nineteen bezel-mounted buttons with configurable text and background colors is in use (See clients in Figure 2.1 for button layout). Based on a survey of vendors, the display chosen was a 10.4 inch Liquid Crystal Display (LCD). This setup provided sufficient capability to access the system components while allowing all of the buttons to border the top and sides of the display. Seven buttons are arranged horizontally across the top of the display and six are arranged vertically along each side. The top buttons serve as hot keys to each of the craft's main systems while the side buttons change function depending on the currently active display.

This setup gives immediate access to any of the craft's main systems through the static buttons along the top. It also provides an interface reminiscent of pull-down menus common in today's applications. In this case, the top buttons serve as the menu bar while the left and right buttons become the menus associated with each top button. There is also a flavor of Web pages associated with the interface in that pushing a top row button will immediately link you to the page associated with it. In many cases, the left and right buttons will effectively become links to different pages within a particular system. Given these similarities to existing desktop systems and to cockpit display systems in both aircraft and land vehicles, the IBS interface had a high probability of being intuitive.

It is also important to note that with large buttons mounted directly on the display it is possible to brace one's hands against the display and use thumbs to manipulate the controls, helping to minimize mistaken inputs due to shocks to the craft. Additional benefits of a configurable hardware button interface is that the user receives tactile feedback when a button is pressed and no screen space is wasted displaying the button texts.

3.3 GUI Models

This project followed the three models for development of a user interface described in [9] – a user's conceptual model, a programmer's model, and a designer's model. In this

frame of reference, a model is defined as "a descriptive representation of a person's conceptual and operational understanding of something." The user's conceptual model is a mental model the user has of the interactions and relationships involved in the system. The user thinks about the system in terms of the tasks it can do and the results it can achieve. The programmer's model is more explicit, coming from the perspective of the person who has to write the system. The programmer must take into account issues such as the platform, operating system, and code. The designer is the architect of the system and is most concerned with visual representations (the "look" of the interface) and interaction techniques (the "feel" of the interface). His model draws from both the user's conceptual model and the programmer's model. Although there is a complete separation between the user and the programmer, the designer is influenced by both of their models (See Figure 3.2).

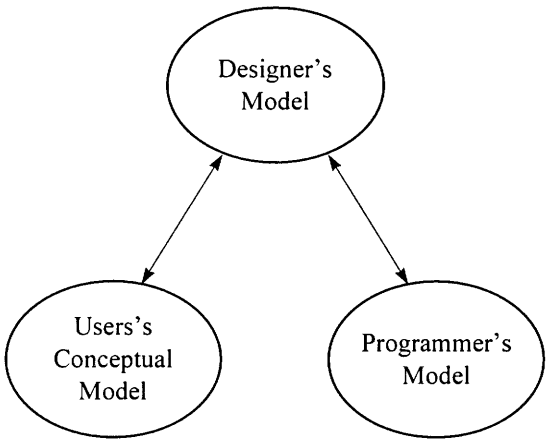

Figure 3.2 - The Relationship among the Models of User Interfaces.

The representations and mechanisms of the designer's model for IBS are described in Chapter 5. The underlying GUI framework, the focus of the programmer's model, possessed its own set of requirements which are described in Chapter 4.

Chapter 4 - GUI Development - Programmer's Model

The programmer's model, the model of the system from the perspective of the person who has to write the code, implements the representations and interactions of the designer's model. The objects and details of this model, although transparent to the end user, determine the overall capabilities of the system. As a result, the programmer's model for IBS entailed a number of requirements.

4.1 Requirements

The IBS GUI framework included requirements for the platform and programming language used as well as provisions for extensibility, mutability, distributed development, and a well-defined interface that could be learned quickly.

The platform chosen for all computers in IBS was a Pentium-based PC running Microsoft Windows NT 4.0. This platform was used for both the clients and the server. Additionally, Microsoft Visual C++ was chosen for all software written for the project providing an object-oriented structure for the implementation.

The largest architectural requirement for the GUI was that it be arranged as a collection of separate processes that are launched as necessary, providing for a complete

separation of dependency. If any particular process were to fail, the other processes would be unaffected. Also, since each process is only concerned with the data specific to that particular system, the system specific functionality of the left and right side buttons is easily encapsulated within each process.

The structuring of the GUI as a set of independent processes necessitated a central dispatcher to manage them. The Client Manager was created to serve as this central dispatcher, providing methods for launching and terminating processes and for communicating input information to them. In creating a well-defined interface for these processes, issues of extensibility and distributed development were also addressed because additions to the GUI can be made by following the proscribed interface. Developer's can create GUI processes with no knowledge of other GUI programs – which also means that additional processes can be developed in the future and seamlessly integrated into the current GUI. The Client Manager also provides for extensibility in terms of new input devices.

The final requirement of mutability was addressed by the Display Wizard, an auxiliary tool for designing GUI screens. With this tool, the system does not need to be recompiled to modify the attributes of individual instruments or the general layout of the screens.

4.2 The Client Manager

4.2.1 Overview

In order to have all of the boat systems developed as separate processes, a central dispatcher was needed to interface with the hardware buttons and to send appropriate messages to the currently active system process (See Figure 4.1). That process could then interpret the messages to carry out the functions that are associated with its left and right side buttons. The dispatcher would also be responsible for managing the execution and termination of the system processes when appropriate. This overall governing component of the GUI is called the Client Manager because each of these individual processes can be viewed as clients within the GUI.

The Client Manager provides three main services. It can launch client programs that are not currently running, it allows the user to switch between client programs seamlessly, and it forwards input messages to the currently active client for interpretation within its scope. All of this is accomplished through an interface that all Client programs are required to follow.

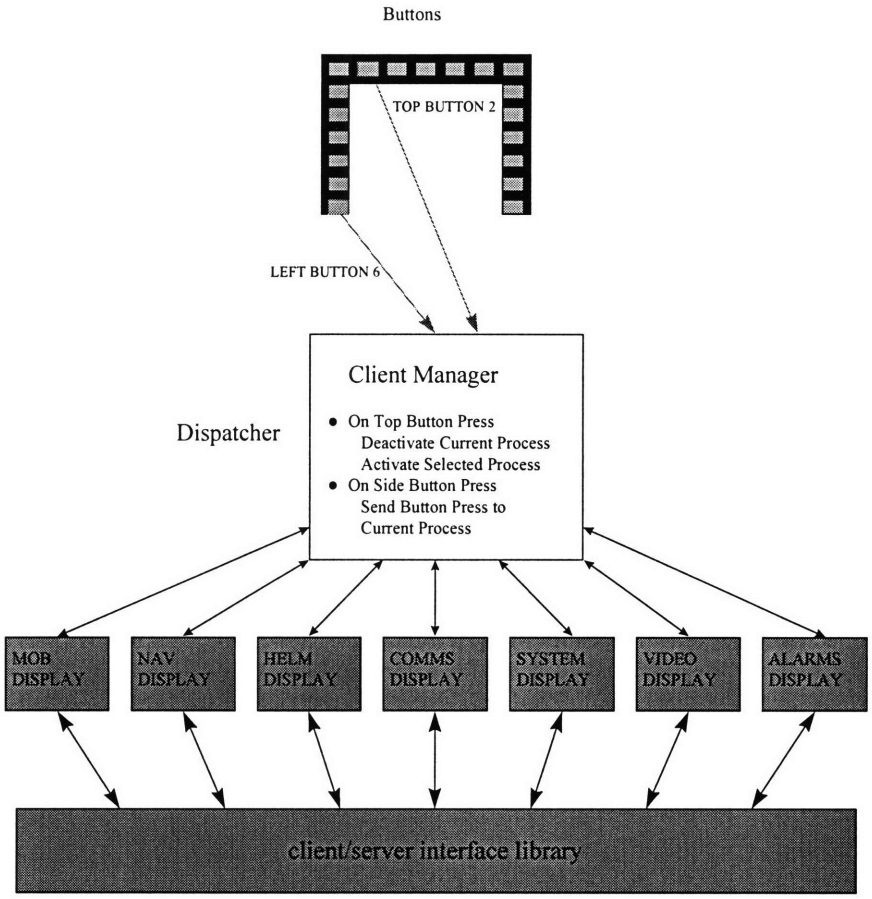

Figure 4.1 - Overall IBS GUI Design.

In order for a particular Client program to be launched, several things must be done. First, the Client must be specified with a system name and executable name in a data file read by the Client Manager on startup. The system name is placed on one the seven top row buttons and when that button is pressed, the executable specified is launched. Next, when the Client program initiates, it must register itself with the Client Manager. This gives the Client Manager a *handle* to the process which it can use to post messages.

In order to switch between multiple Clients, the Client Manager keeps track of which Client program has focus. The Client that is currently active and visible to the user is considered to have focus while those hidden do not. When a Client is launched it is given focus and only one Client has focus at any given time. After that point, a Client gains focus whenever the top row button associated with it is pressed. At the same time, the program that last had focus, loses it. All of this is accomplished through two messages, *LoseFocus*, which is posted to the currently active Client, and *GainFocus*, posted to the Client associated with the top row button pressed. Upon receiving these messages, it is the responsibility of the Client to iconify or restore itself as appropriate. There are also two corresponding functions for acknowledging these messages that the Client should call after taking these actions.

The Client Manager also sends a *ButtonHit* message to the Client that currently has focus whenever one of the side buttons is pressed. A Client will only receive this message if it has focus. The *ButtonHit* message specifies a parameter indicating the ID of the button pressed. The Client is then responsible for taking appropriate action based on this message.

The last message a Client can receive is a *ShutDown* message. When the Client Manager exits, it will send this message to all registered Client programs. Each Client should exit gracefully upon receiving this message.

There are several other functions available to the Client programs through the Client Manager interface. Each Client has the ability to change the text that appears on the left and right side buttons. Additionally each Client has a local stack of button texts that they can maintain. For example, a common practice is for a Client to set the texts on

left and right side buttons when the Client is launched and then push its button text stack, thereby saving the texts, when a *LoseFocus* message is received. When the client receives a *GainFocus* message, it can pop its button text stack. This will restore the pushed texts to the buttons. Actions such as pushing or popping button text stacks should be taken before a *LoseFocus* or *GainFocus* message is acknowledged.

4.2.2 Input Devices

During development, on-screen software buttons were used to mimic the eventual behavior of hardware buttons. This allowed button hits to be registered through the normal CButton Microsoft Foundation Class [20]. In order to facilitate the use of external input devices, the interface to the client manager also allows a client to emulate a button hit. Since any hardware buttons added to the system will require a low level driver, this driver can simply call the button hit emulation function when appropriate. Although the nineteen button interface is fixed, any input device can be used as a front end to this interface including hardware buttons or speech recognition.

4.3 The GUI Clients

The individual Client programs consist of a base dialog window, hereafter referred to as the Client Shell, a set of Display windows, and a set of instruments. Displays, contained within the Client Shell, are the separate pages of information within the particular system. Only one Display, maximized within the Client Shell, is visible at any given time. Each Display then contains groups of instruments for a specific page (See Figure 4.2).

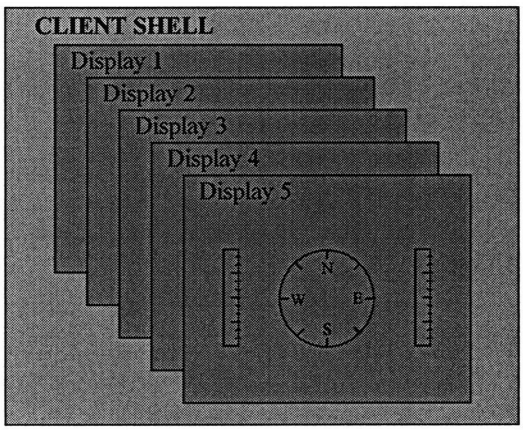

Figure 4.2 - IBS GUI Client Structure

This structure for the Clients provides a separation of functionality among the elements that comprise them. The Client Shell is at the top level and handles all of overall Client responsibilities, such as handling messages received from the Client Manager. A Display serves as a container for the instruments that are part of the

particular boat system, keeping the instruments grouped according to function. Finally, all of the functionality necessary for drawing and updating instruments is encapsulated within the instruments themselves. This multi-level design helps to minimize the amount of work necessary to create additional Clients and Displays.

4.3.1 Base IBS Dialog Class - IBSClientDlg

The Client Shell for each Client is derived from a base dialog window class, IBSClntDlg. In effect, the Client Shell's relationship with the Displays is similar to the Client Manager's relationship with the Client in that it controls which Display is currently visible and what information that Display receives. In addition, the Client Shell is the container for all of the Client's Displays and attends to all of the Client's overall responsibilities. It handles messages received from the Client Manager, obtains data from the Server necessary for drawing its Displays, and executes appropriate button press functions.

There are four messages from the Client Manager, described in the Client Manager section, that the Client Shell must handle: *LoseFocus*, *GainFocus*, *ButtonPress*, and *ShutDown*. When the Client program receives a *LoseFocus* message, the Client Shell pushes its button text stack, iconifies itself, and then acknowledges the *LoseFocus* message. Similarly, when it receives a *GainFocus* message, the Client Shell pops its button text stack, restores itself, and acknowledges. A *ButtonPress* message results in a

call to an analogous button press function. The Client Shell has functions for each of the left and right side buttons. These button functions are overridden in the child classes derived for each Client program so as to provide Client specific functionality. The *ShutDown* message causes the Client Shell to exit and the Client program to terminate.

Data is generally obtained from the Server by the Client Shell through a polling routine which is launched as a separate thread. This is done at the Client Shell level rather than the Display or instrument level so as to minimize repeated calls to the Server. At a specified frequency, determined by a developer-defined wait time in milliseconds, the thread calls an Update function, overridable in the derived classes. This Update function calls the appropriate interface library functions (See Chapter 3) and does any parsing of the returned data that may be necessary. Each time the Server is polled, the Client Shell instructs the currently visible Display to update itself. Each instrument within the currently visible Display is then responsible for redrawing itself based on the new data.

4.3.2 Display Class

All of the instruments for the IBS GUI are drawn using OpenGL drawing commands [2]. The Display Class is a window class that supports such commands. It's only purpose is to serve as a container for the groups of instruments. As a derivative of the base window class, it has functions for the Client Shell to hide and show a Display as necessary.

4.3.3 Instrumentation

All instruments are derived from a base instrument class that provides for common traits and functionality. Common to all instruments are traits such as location (x and y screen coordinates), scale, and color. The base instrument class also has a Data Pointer that points to the variable (hereafter referred to as the Data Variable) containing the data that the particular instrument is monitoring. This parameter, which in most cases determines an instrument's state, can be of any basic type (integer, float, etc.) or an array of integers, characters, or strings.

The base instrument class also includes a number of general functions necessary for all instrument types. These include functions for creation, initialization, modification, data input, and drawing. All but Create are implemented as virtual functions, called from the base class as necessary. Instrument specific code that will be carried out automatically can be written for any derived classes. The Draw functions for the derived instruments make use of OpenGL drawing commands [2].

The library of instruments derived from this base class is shown in Figure 4.3 and described in the sections below.

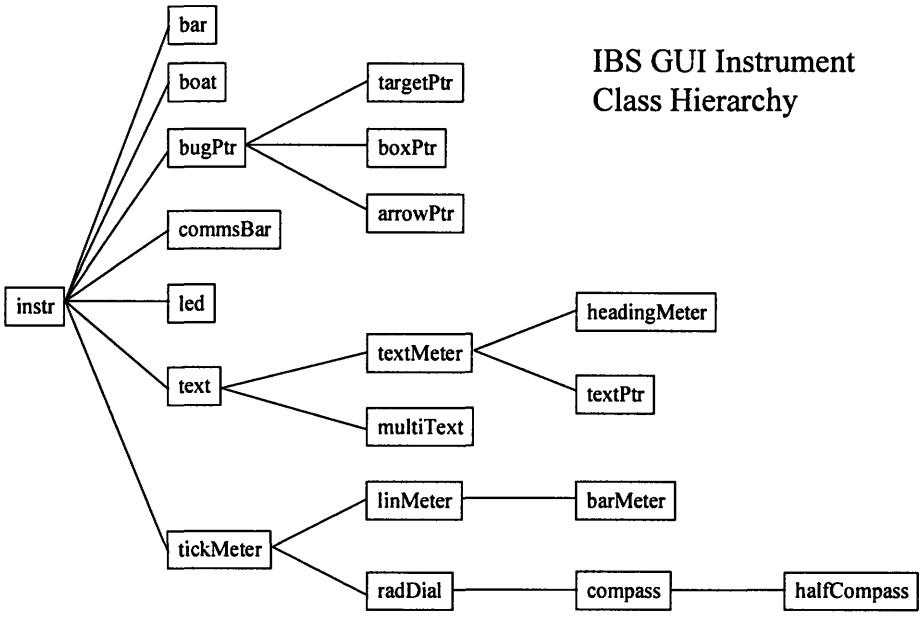

Figure 4.3 - IBS GUI Instrument Class Hierarchy

4.3.3.1 Static Text Class

The static text class allows for the display of a piece of static text in a Display. The class provides for labels, messages, and on screen instructions.

4.3.3.2 MultiText Class

A MultiText is an extension of the text class that allows a piece of text to change based on an index variable. An array of possible texts is specified and the Data Variable is an integer that specifies which text in the array is to be displayed.

4.3.3.3 Text Meter and Text Pointer Classes

A Text Meter is an item of text that is tied directly to a piece of data. The Data Variable for this instrument type can be of any basic data type. The value stored in this variable will be written as text, based on a format string specified in the same manner as for the standard ANSI C "printf" function.

A Text Pointer is simply a Text Meter that moves over a specified distance as its Data Variable changes. While the Text Meter can be optionally boxed, the Text Pointer is automatically boxed with a triangle pointing left or right as desired (See Figure 4.4 at the end of the chapter).

4.3.3.4 Linear Meter and Bar Meter Classes

A Linear Meter is a gauge whose physical representation is a box, oriented horizontally or vertically, that is used to display a single parameter on a linear scale. A Bar Meter (a basic bar graph), derived from the Linear Meter class (See Figure 4.5), is the most utilized instrument for this application. A Bar Meter allows for a number of customizations including specifications for range of data and critical zones and customizable tick marks. The size of the bar is determined by the Data Variable.

4.3.3.5 Radial Dial and Compass Classes

The Radial Dial class provides for the display of information in radial style gauges. In this sense the gauge would be static with some type of indicator that marked the current value of the data. The two compass instruments developed for IBS – full and half rosettes (See Figures 4.6 and 4.7) – are extensions of this class. In addition to an indicator, which for a magnetic heading compass indicates the current course to steer, the gauges themselves rotate based on current heading.

4.3.3.6 LED Class

The LED class is useful for any data with a binary state, for example on or off. It can be configured to display any color for each of the two states. It also provides several possible icons for the LED including a general circular LED and several IBS specific icons (See Figure 4.8).

4.3.3.7 Bug/Pointer Class

The bugPtr class offers a base class for creating any type of pointing object and have certain attributes such as a focus point (See Figure 4.9). The derived classes are required to specify how the pointer is drawn. The arrowPtr class is an example of such a derived class that is drawn as an arrow that can be used to point out an area of interest.

4.3.3.8 Other Instrument Classes

Several other classes of instruments have been developed to address IBS specific needs. Among these are a class for drawing the outline of particular boat used to show relative location of lights and bilge pumps and the bar class that is used to draw the bar in a Bar Meter.

4.3.4 The Display Wizard

Although it is possible to create and then freeze a specification for other applications, it is impossible to form a "stable" specification for a user interface. There is simply no complete checklist of rules [6]. Therefore, a GUI should be mutable in that it should allow any changes to any non-critical aspect of the display (i.e. colors, layouts, etc.) that may arise through user testing and feedback. Although incorporating mutability requires a larger initial time commitment, it accelerates the development process because the design can be continually reviewed and modified based on, for example, user community feedback. This versatility was necessary in IBS for modifications to the appearances of instruments to be easily accomplished. In order to meet this goal, almost every aspect of each instrument class is represented by a member variable that can be modified to obtain a different look or effect.

The management of such a large set of configuration variables becomes increasingly difficult. If the developer wishes to change the color of the bar in a Bar Meter from blue to green to see how it looks, finding the appropriate variable in the code and recompiling the system just to see this simple change is tremendously time consuming. To simplify this process an auxiliary development tool was created – the Display Wizard. The Display Wizard allows the developer to design and build a display, placing all of the instruments and text as desired. It allows the developer to add as many

instruments as she wishes, configure and manipulate them, or delete them. Once the developer is satisfied with the display, she can save the information to a data file. This data file can then be loaded into the IBS GUI for the appropriate display which will be automatically reconfigured at the next start up.

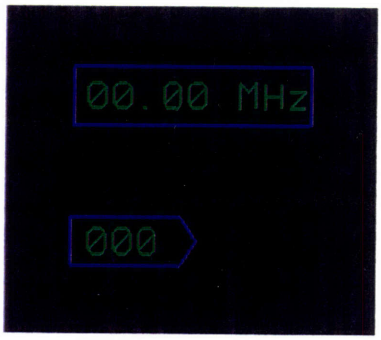

Figure 4.4 - Text Meter and Text Pointer

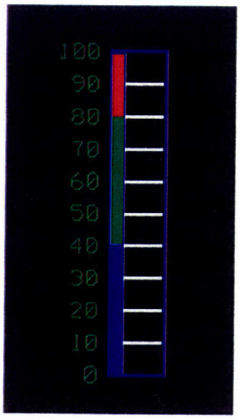

Figure 4.5 - Bar Meter

Figure 4.6 - Compass Rosette showing heading (38°) and desired heading (64°)

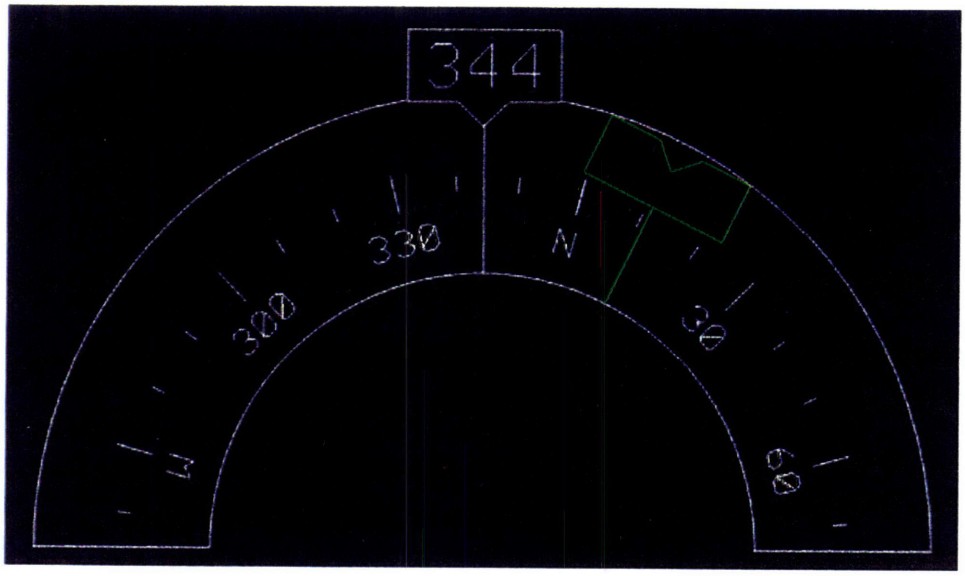

Figure 4.7 - Half Rosette showing heading (344°) and desired heading (11°)

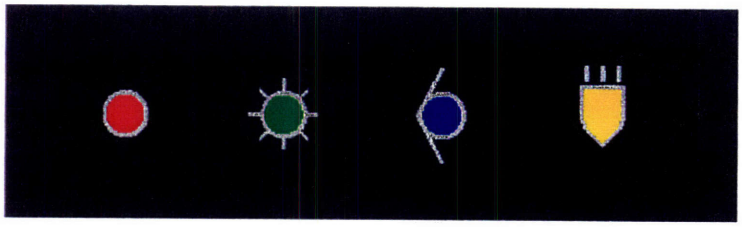

Figure 4.8 - Simple, Compass, Nav Light, and Spot Light LEDs

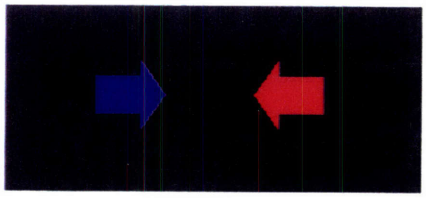

Figure 4.9 - Arrow Pointers

Chapter 5 - GUI Development - Designer's Model

The designer is mostly concerned with the visual representations and interaction mechanisms of the interface. He is influenced by both the user's conceptual model and the programmer's model and uses them to create an effective interface [9]. The user, however, is often unable to describe their conceptual model, even when they are directly involved in the design process as was the case in this project. General principles for user interface design serve as guidelines when this is the case, and have been followed for the IBS GUI. In some cases, tradeoffs were necessary due to the specific nature of the interface developed. These cases are discussed in detail in the following sections.

5.1 Requirements

Many of the design goals common among graphical user interfaces also apply to the IBS GUI and are described below [12]. The manner in which each of these principles is addressed in the IBS GUI is described in the next section.

- Clarity – The interface must be clear in visual appearance and words and text should be unambiguous and simple.

- Comprehensibility – The interface should be intuitive, flowing in a meaningful order. Steps to complete a task should be obvious and predictable.
- Consistency – Similar tasks and representations should be consistent throughout the interface. Unnecessary variety requires more training time, more specialized knowledge, and more frequent changes in procedure. A consistent interface will encourage the development of behavior patterns [11].
- Control – The user should feel that they are in charge and that the system is responding to their actions rather than the other way around.
- Efficiency – Eye and hand movements should not be wasted. The user's attention should be captured by relevant elements when appropriate.
- Forgiving – People will make mistakes, the system should tolerate these errors.
- Simplicity – Never include unnecessary complexity.

5.2 Implemented Clients

Several standards and guidelines were established for the implemented GUI Clients to address the issues mentioned in the previous section. These include standards for button functionality, inter-process consistency, and the display and acquisition of data.

The functionality of the buttons is established as follows: The top row buttons serve as menus for each of the GUI Clients. Within each Client, the left buttons serve as subsystem menus and the right buttons serve as functions within these subsystems. This paradigm contributes to both the consistency and the comprehensibility of the system because with any GUI Client the interface is the same. The user can quickly become attuned to this interface and begin to accomplish tasks intuitively. This interface also puts the user in control, allowing them to switch between the GUI Clients at any time, regardless of his position or state within the currently focused Client.

Having only nineteen buttons, a result of the limited console space available for mounting them, necessitated the use of modes in some cases, described in further detail in the sections on the Clients that contain them. Modes are states in which only a limited set of functionality is available to the user. They are generally considered to detract from the consistency – and therefore from the usability – of a system [12]. While using modes breaks the consistency of the interface, the tradeoff in this case is that a *three button press* limit for data acquisition can be maintained throughout the system.

The *three button press* rule was followed in order to maintain the efficiency of the system, allowing the user to access data quickly. This means that it only takes three button presses to get to any data within the IBS GUI. Entering data into the system may require more than three button hits since there is often a confirmation button to prevent mistaken inputs. This will be explained further in the sections about the individual Clients.

Efficiency is also improved through the use different font attributes. Data is displayed in a bold font while less important information such as labels and units are

displayed in a lighter font. This allows the user to focus on the pertinent data quickly without being distracted by the auxiliary information.

In order to maintain consistency as the user switches from Client to Client, each Client remains in the state in which the user left it. The exception to this rule is that a Client will not remain in a modal state, instead reverting to the standard interface. Otherwise, if a Client were left in a modal state and the user returned to that Client after some amount of time he would have to exit that modal state to be able to use the standard interface. This method helps to alleviate user confusion.

Four GUI Clients have been developed with the framework described in Chapter 4 and these design guidelines. The Clients include an interface to the communication systems, a Client that incorporates all of the information necessary for the driver, a monitor for general boat systems, and a Client for handling alarms. These Clients, together with an integrated navigation package – developed separately – comprise the basic set of functionality necessary to operate the craft. The Figures of the different Clients are located at the end of the Chapter. These images contain on-screen buttons that were used during development to emulate the eventual hardware buttons.

5.2.1 COMMS Client

The COMMS Client allows the user to control any of the radios that are available on the vessel. The available radios are displayed across the top of the screen and the user can

toggle through them using the left "Select" button (See Figures 5.1 - 5.5). A green box highlights the currently selected radio. The attributes for each radio maintained on the server include transmit and receive frequencies, squelch, volume, output power, power, encryption, and modulation.

Transmit and receive frequencies, which can be controlled separately in some radios while in others they are tied together, can be modified in two ways. Selecting "Presets" from the left buttons causes a set of preset frequencies (in MHz) to appear on the right buttons (See Figure 5.3). A "Page Down" key on the last right button allows for a second page of presets or ten in all. Pushing one of these preset buttons causes the transmit and receive frequencies to be set to the frequencies listed on the button. Since these presets are maintained in software, *all radios can have presets regardless of whether the hardware supports such a feature*. The second manner in which frequencies can be modified is to manually set the transmit and/or receive frequencies to a specific value. This is accomplished by first pressing the left "Set Freqs" button (See Figure 5.4). This brings up buttons on the right for manually setting frequencies and for setting up preset values. The top right button toggles through the options for which frequency the user wishes to set - transmit, receive, or both. The second right button brings changes to a data entry mode where the user can type the new frequency (See Figure 5.5). A separate mode is needed in order to have ten buttons for entering the digits 0-9 – displayed on the first five buttons on the left and right – a "Cancel" button on the bottom left button, and an "OK" button on the bottom right button. The interface is similar to a Bank ATM in that as the digits are pressed, the numbers scroll to the left. If the value specified is valid for the selected radio, the frequency will be set to this frequency and the

mode ended when the user presses "OK." Presets are set in a similar manner using the bottom three right buttons. The only additional action necessary is for the user to choose which preset will be modified by toggling through the them with the fifth right button.

Volume, squelch, and output power are represented by the three bar graphs beneath the frequencies. These can be modified by first pushing the left "Volume" button and manipulating the up and down buttons for each attribute that appear on the right (See Figure 5.2).

The COMMS interface also incorporates controls for individual radio power, encryption, and modulation. Pressing the last left button labeled "Control" brings up controls on the right for each of these attributes (See Figure 5.1). Power and encryption are two-position switches while the modulation button allows the user to toggle through all of the possible modulations for the selected radio.

Finally, the COMMS Client allows the user to specify up to five radios to which he will listen and one radio through which he will talk. The red arrow points to the radio that is currently selected for talking and the left "PTT" button, or Push to Talk, toggles through the available radios. To monitor a radio, the user presses the left "Control" button and then the "Monitor" button on the right. This toggles monitoring on and off for the currently selected radio. Up to five different radios may be monitored at a time.

5.2.2 HELM Client

The HELM Client provides all of the monitoring information necessary for the driver of the vessel. It consists of three screens: a full compass rosette, a half rosette, and a screen for monitoring drive and trim tabs (See Figures 5.6 - 5.11). The full and half rosette pages display essentially the same information in two different formats.

In addition to current heading, each of the compass pages also contains other pertinent navigation information. Each compass includes reciprocal heading – 180 degrees opposite current heading – a green course-to-steer indicator, and two blue arrows indicating direction to steer. The left side of these pages holds information about the current waypoint or target. The upper left corner contains the ID for the current waypoint and its latitude and longitude. The lower left displays cross track error (XTE), bearing to waypoint (BTW), distance to waypoint (DTW), and course to steer (CTS). The right side of the page displays status information for the vessel. In the upper right is the current time and in the lower right are the vessel's latitude and longitude, its speed over ground (SOG), and its course over ground (COG). All of the data displayed is obtained from the server which is responsible for maintaining this information.

The drive and trim tabs page offers an abbreviated set of information providing only the status information that is shown on the right side of the compass displays.

5.2.3 SYSTEM Client

The SYSTEM Client incorporates most of the boat specific systems. This Client will vary from vessel to vessel to reflect each boat's configurations. The current SYSTEM Client is tailored to the HSAC, which has two engines, starboard and port, and two fuel tanks, forward and aft, for each engine.

Three engines status pages provide all of the information regarding the state of these components. The first page, available by pressing the left "Engine" button and the right "Rpm" button shows the rpms for each of the engines as well as the drive and trim tabs that were included in the HELM Client (See Figures 5.12 and 5.13). Additionally, each engine has a binary EFI, or Electronic Fuel Injection, indicator.

The other two Engine pages display information regarding water, oil, and transmission fluid temperatures and pressures. These pages are accessed by pressing the left "Engine" button and then either the right "Temp" button for temperatures or the right "Press" button for pressures (See Figures 5.14 - 5.17). Each of the pages is divided into port engine information on the top and starboard engine information on the bottom. The EFI indicators are also displayed on these pages.

Information regarding the fuel level is available by pressing the left "Fuel" button (See Figures 5.18 and 5.19). This page is also divided into port engine information on top and starboard engine information on the bottom. Fuel level – shown as a percentage

– for both forward and aft tanks, fuel pressure, and rate of fuel flow are listed for each engine. This page also provides two buttons on the right for toggling on and off the primary and secondary fuel pumps.

Pressing the left "Elect" button brings up the status for the port and starboard batteries and inverters (See Figures 5.20 and 5.21).

The left "Bilge" button brings up an icon of the vessel with three buttons on the right for controlling the forward, cockpit, and engine bilge pumps (See Figure 5.22). By default these pumps are in "auto" mode meaning that they should turn on as necessary. Pressing the right buttons manually turns on the pumps. Pressing the right buttons again switches the pumps back into "auto" mode.

The left "Lights" and "Control" buttons deserve special mention because these are two cases where the physical limitation of nineteen buttons necessitated the use of modes. The Lights page allows the user to turn on and off the light systems on the vessel while the Control page allows the user to toggle any of the electrical relays. The standard interface for these pages would be for the six right buttons to control the lights and relays. In each case, however, there are more than six items that can be toggled. Using scrolling or paging buttons would limit the number of lights or relays that could be displayed at a time to four. As a result, toggling many of the lights and relays would require multiple extra button hits just to find the desired switch. Instead, separate modes for lights and controls were created, allowing these pages utilize both the left and right side buttons to list the lights and relays. Included on each page is a "Done" button to end the mode and take the user back to the standard SYSTEM Client interface.

The Lights page displays two icons of the boat, one on the left for interior lights and one on the right for exterior lights (See Figure 5.23). The boat icons show the true state of the lights on the boat. As the user presses the left and right buttons associated with lights on the ship, the state of the buttons will change verifying the user's selection (See Figure 5.24). This will not change the state of the lights on the boat. Lights will only be toggled when the user presses the right "Commit" button (See Figure 5.25). Conversely, if the user presses the right "Cancel" button, the buttons revert to the state shown in the boat icons. A red "changed" text message pops up over the boat icons if the true state of the lights and the state set by the users does not match, indicating that the user should either commit or cancel their settings. The Lights page is designed in this manner so as to prevent incorrect light selections from jeopardizing mission critical situations.

The Control page (See Figure 5.26) lists ten different power relays that can be turned on or off. As opposed to the "Lights" page, these switches do not require confirmation. The right "Done" button returns to the SYSTEM standard interface and the last left button allows the user to exit the IBS GUI (See Figure 5.27).

5.2.4 ALARMS Client

A set of specified "safe" ranges is maintained on the server. When the server detects that any data has exceeded a specified range, an alarm is signaled. The ALARMS Client polls

for these alarms, signaling to the user that an alarm has occurred by turning the Alarm button in the top row red. Switching to the ALARMS Client, the user sees a list of currently active alarms (See Figure 5.28). Up to six alarms are visible on the screen at a time. If there are more than six alarms active the fifth and six right buttons scroll the list up and down. Pushing one of the left buttons selects the alarm that is opposite that button (See Figure 5.29).

When an alarm has been selected, any of three actions may be taken. The alarm may be canceled, in which case it is removed from the list of active alarms. It may be acknowledged, causing the alarm to be removed from the list and the threshold that was violated, causing the alarm to be signaled, is extended. Finally, the user may choose "Go To." This action brings up the GUI page that contains the monitor for the data that caused the alarm. The indicator of the instrument that is monitoring the critical data turns red when an alarm is signaled.

If the user leaves the ALARMS Client, either by pressing "Go To" or by manually selecting another Client, and there are alarms still active, the Alarm button in the top row turns yellow to remind the user. If a new alarm is signaled, the Alarm button again turns red signifying that a new alarm has been raised.

Because alarms have unique functionality, a few exceptions to the normal operation of a Client had to be made. The Client Manager is designed to create a complete separation of the GUI Clients. This allows the Clients to be developed independently with no knowledge of each other. For the alarm Client to execute a "Go To" command, however, it must have knowledge of the other Clients. Since the Client Manager allows for simulated button hits with the MenuItemHit function, the Server

stores this information as a series of button hits associated with a particular alarm. When the "Go To" button is pressed, a set of button hits is emulated taking the user to the appropriate page in the GUI. This information must be specified for every alarm that can be raised and is maintained on the Server to guarantee system-wide consistency.

In keeping with the idea of completely separate Clients, only the currently focused Client should be able to affect the buttons (i.e. change the button texts, highlight a particular button, etc.). This would guarantee that when a Client is focused, it has complete control and cannot be preempted by the actions of other Clients. The ALARMS Client, however, must be able to change the last top button red when it does not have focus in order to signal an alarm to the user. In order to accommodate this need, unfocused Clients were given the ability to change the state of a button – background color, highlighted or not, and multiline or not – but not the text of a button. This tradeoff, while necessary for the ALARMS Client to function, can cause unwanted effects if used improperly but these effects are limited to button state.

Figure 5.1 - COMMS Client, Control Page

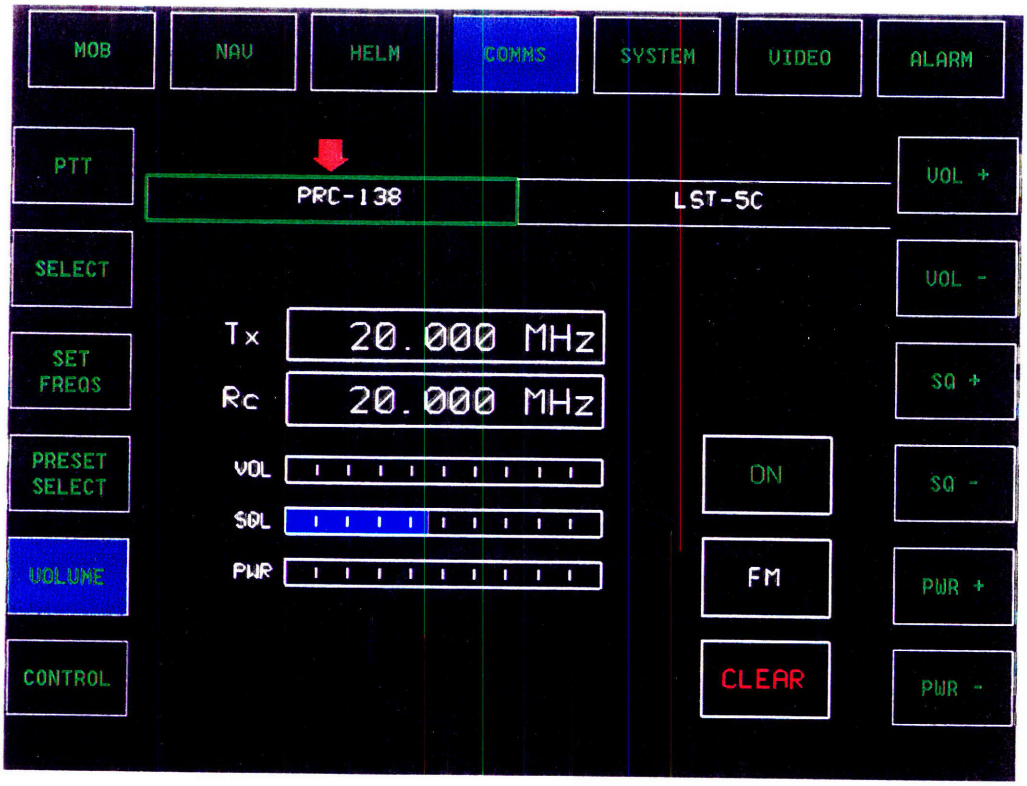

Figure 5.2 - COMMS Client, Volume Page

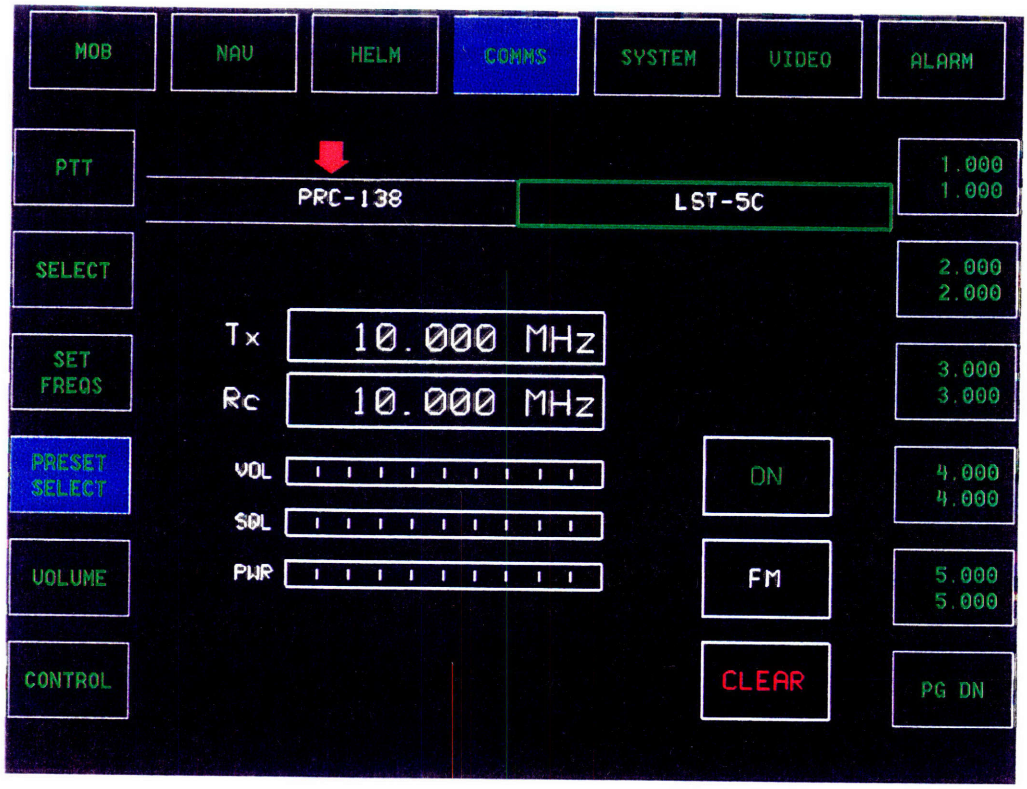

Figure 5.3 - COMMS Client, Preset Select Page

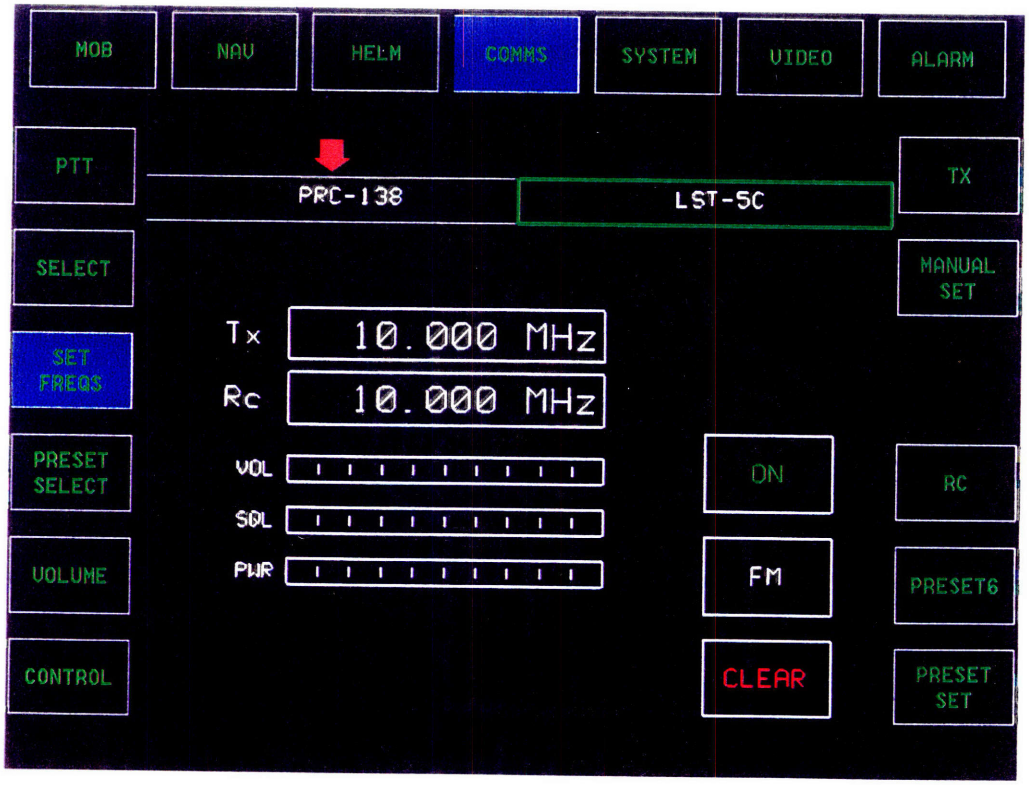

Figure 5.4 - COMMS Client, Set Frequencies Page

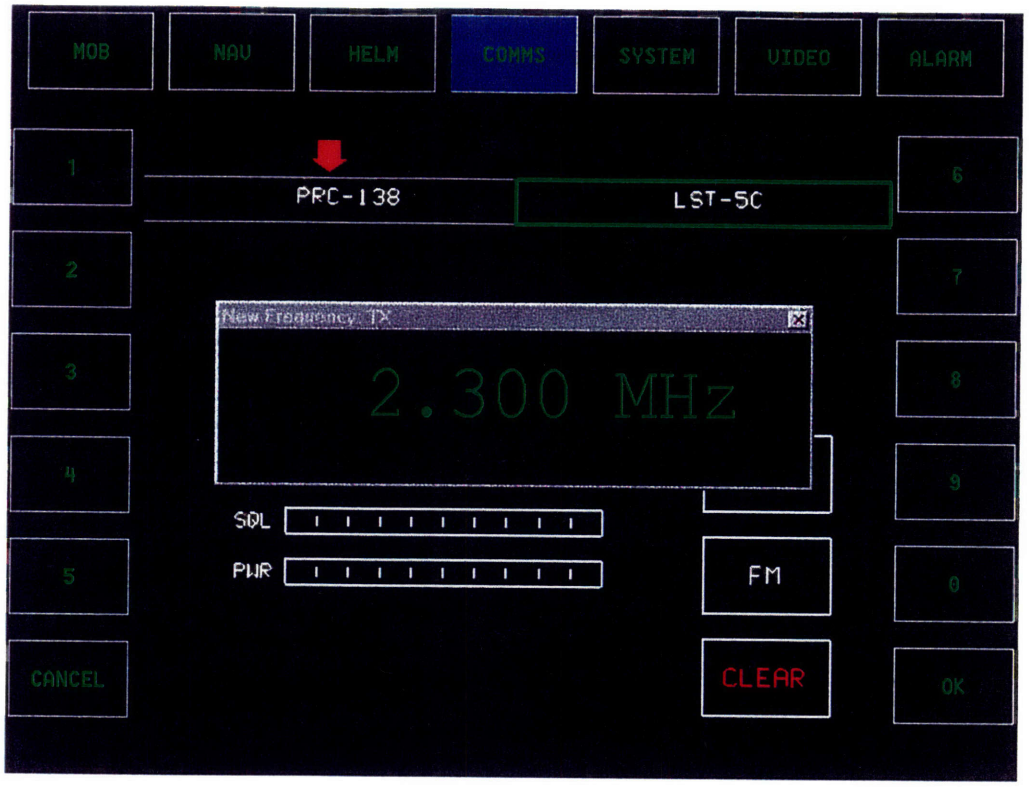

Figure 5.5 - COMMS Client Set Frequencies Dialog

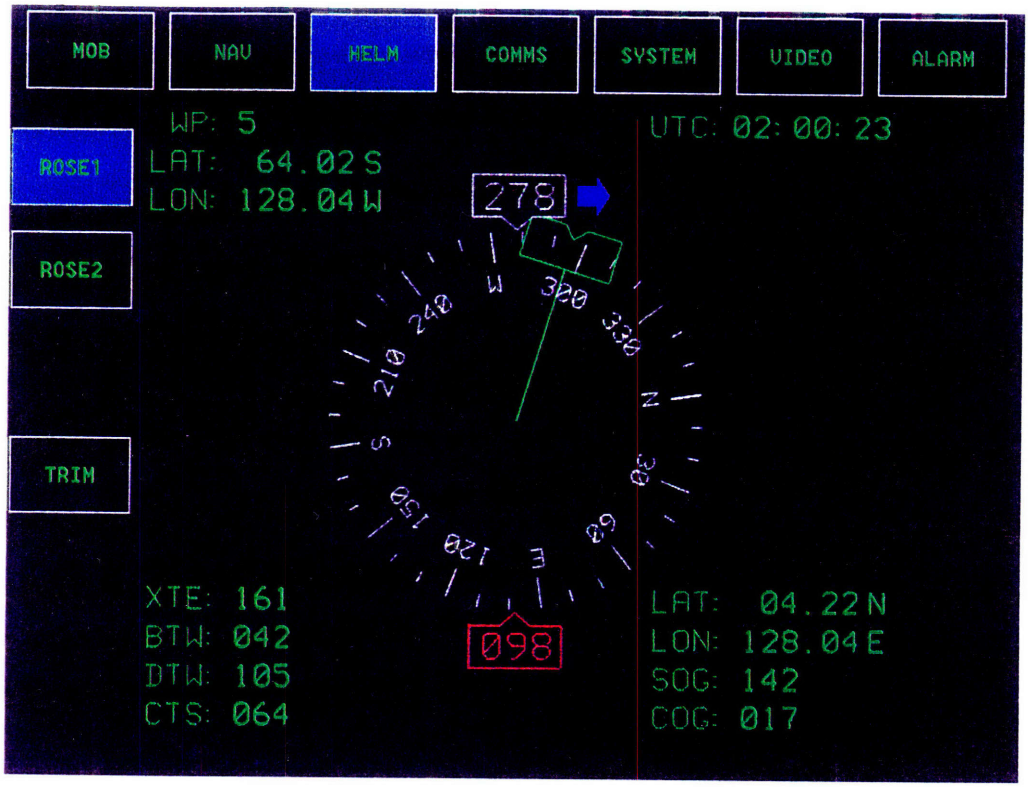

Figure 5.6 - HELM Client, Full Rosette Page #1

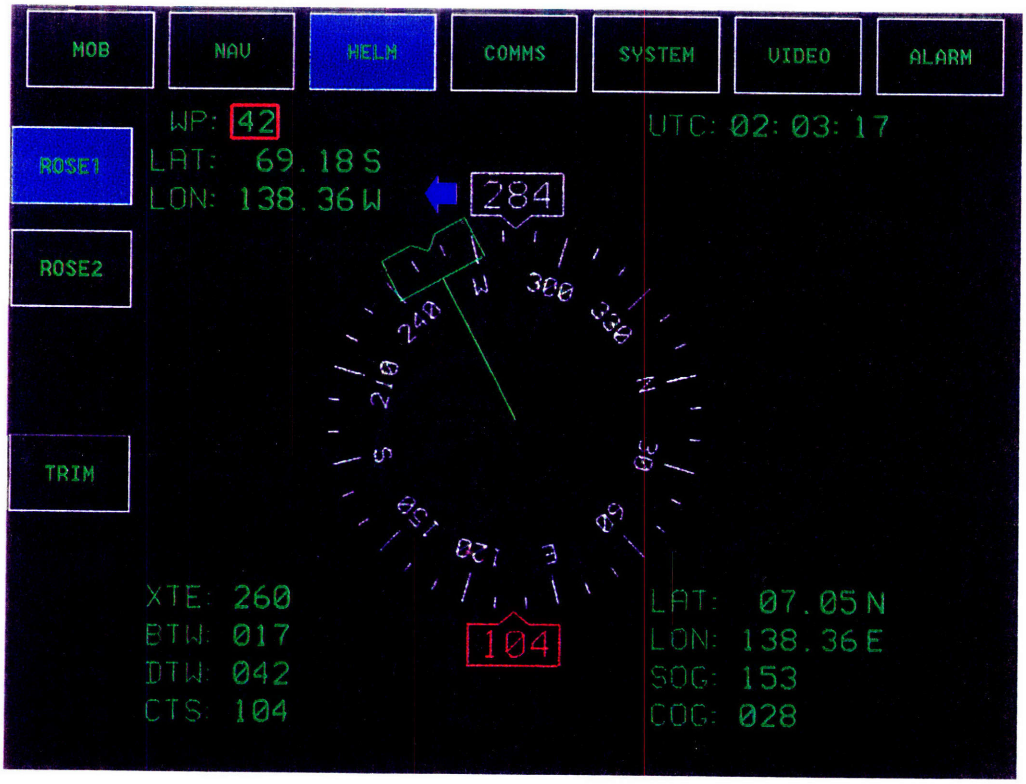

Figure 5.7 - HELM Client, Full Rosette Page #2

Figure 5.8 - HELM Client, Half Rosette #2

Figure 5.9 - HELM Client, Half Rosette Page #2

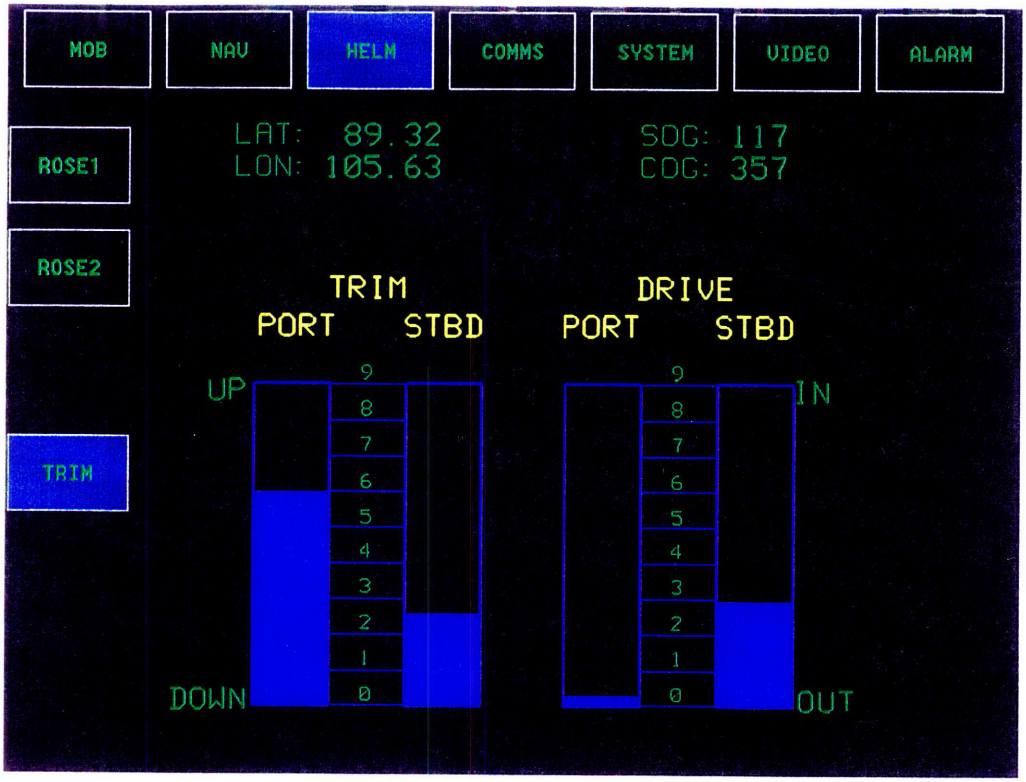

Figure 5.10 - HELM Client, Trim Page #1

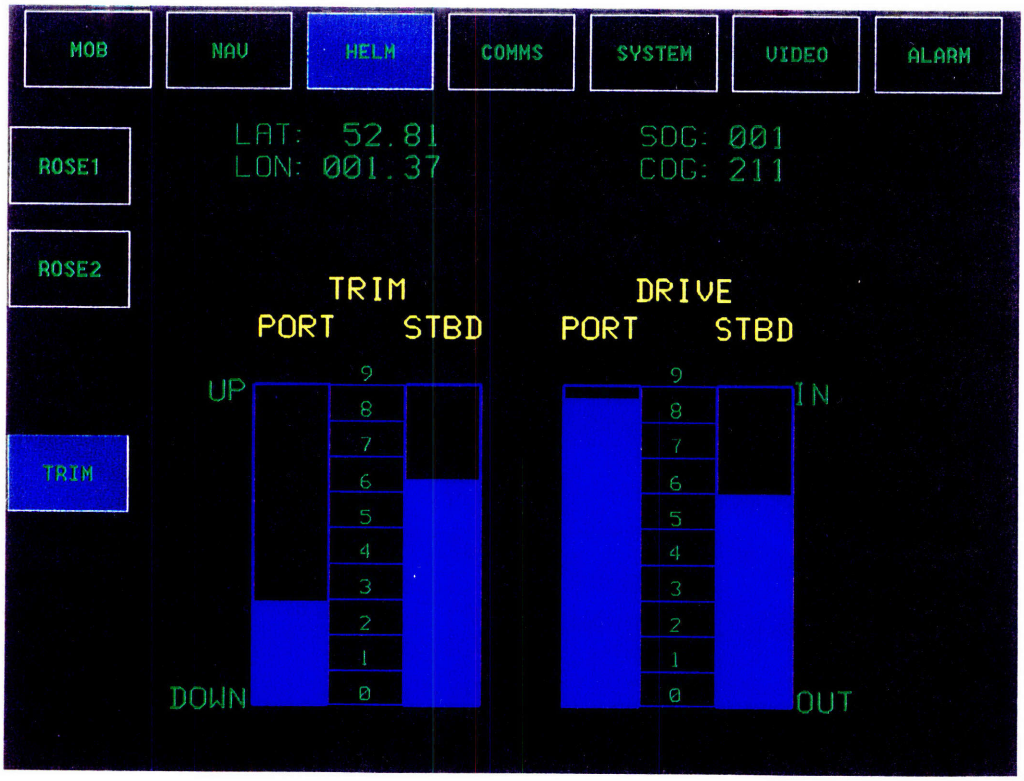

Figure 5.11 - HELM Client, Trim Page #2

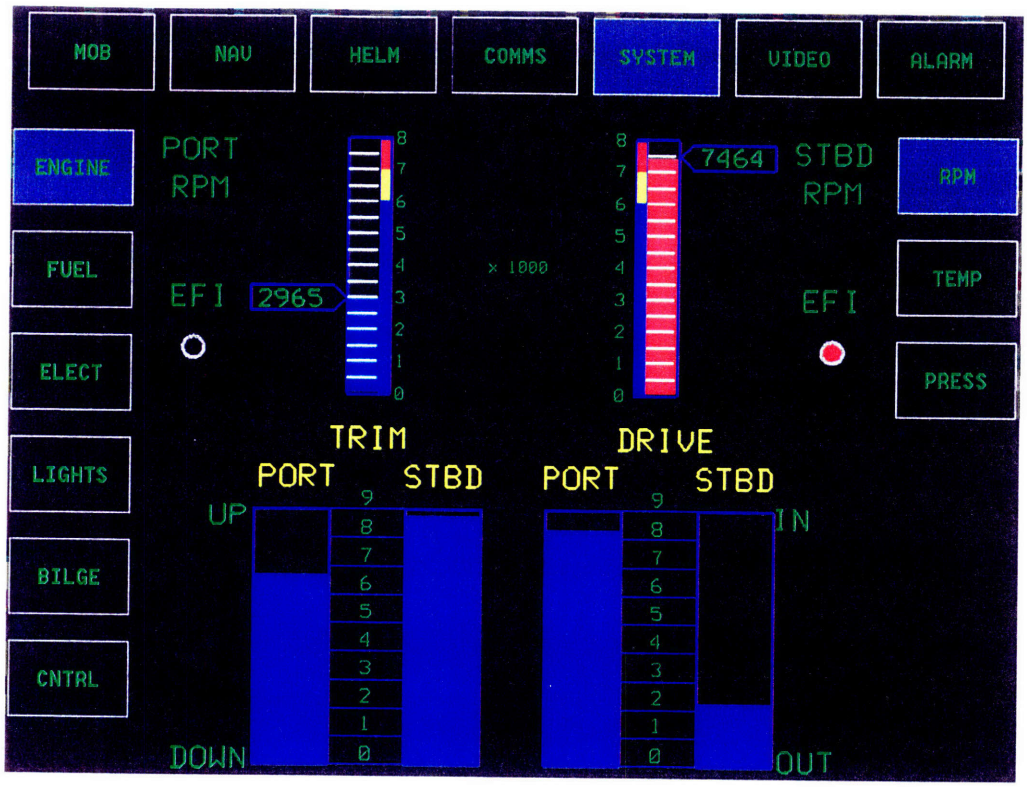

Figure 5.12 - SYSTEM Client, Engine, Rpm #1

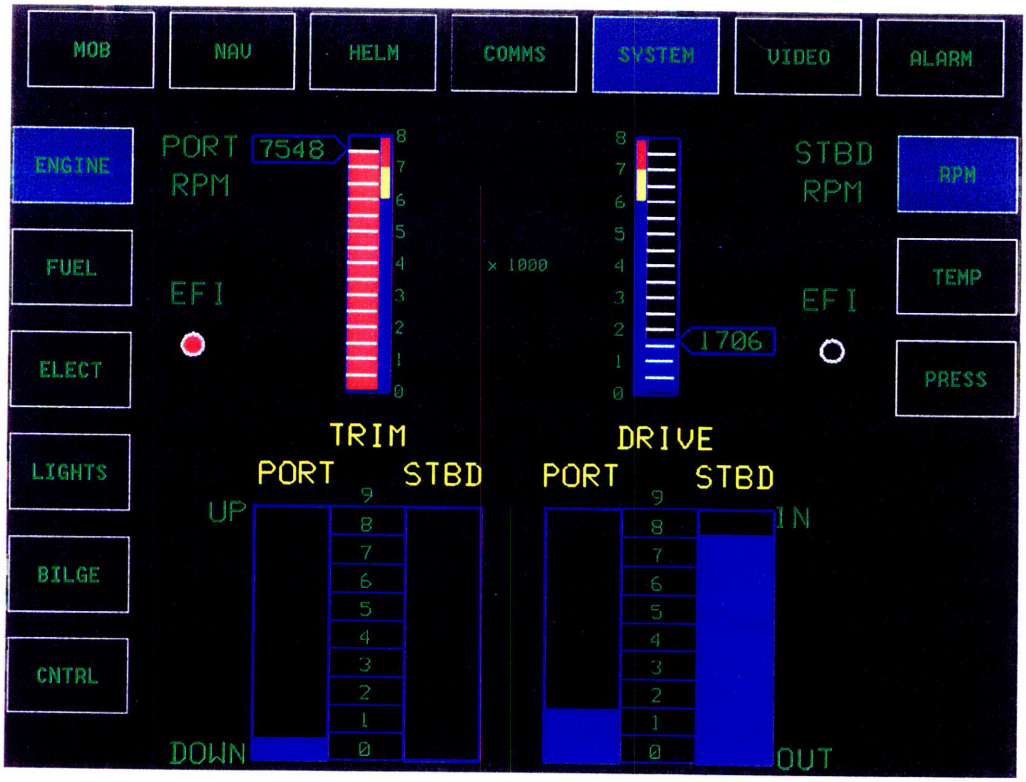

Figure 5.13 - SYSTEM Client, Engine, Rpm #2

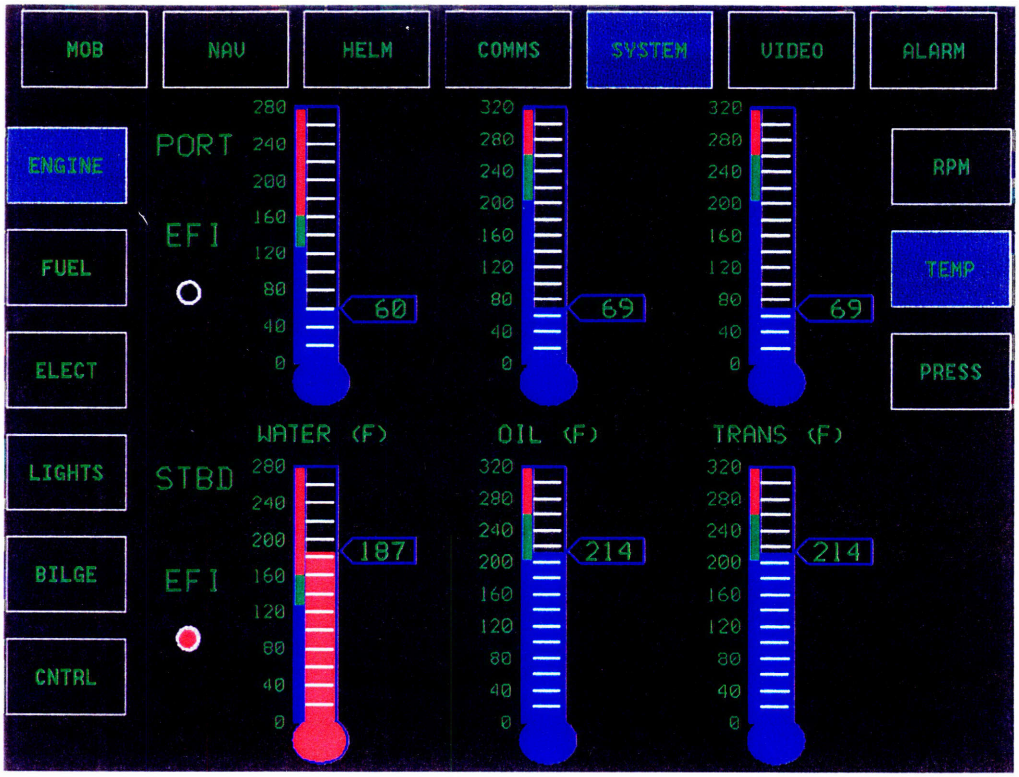

Figure 5.14 - SYSTEM Client, Engine, Temp #1

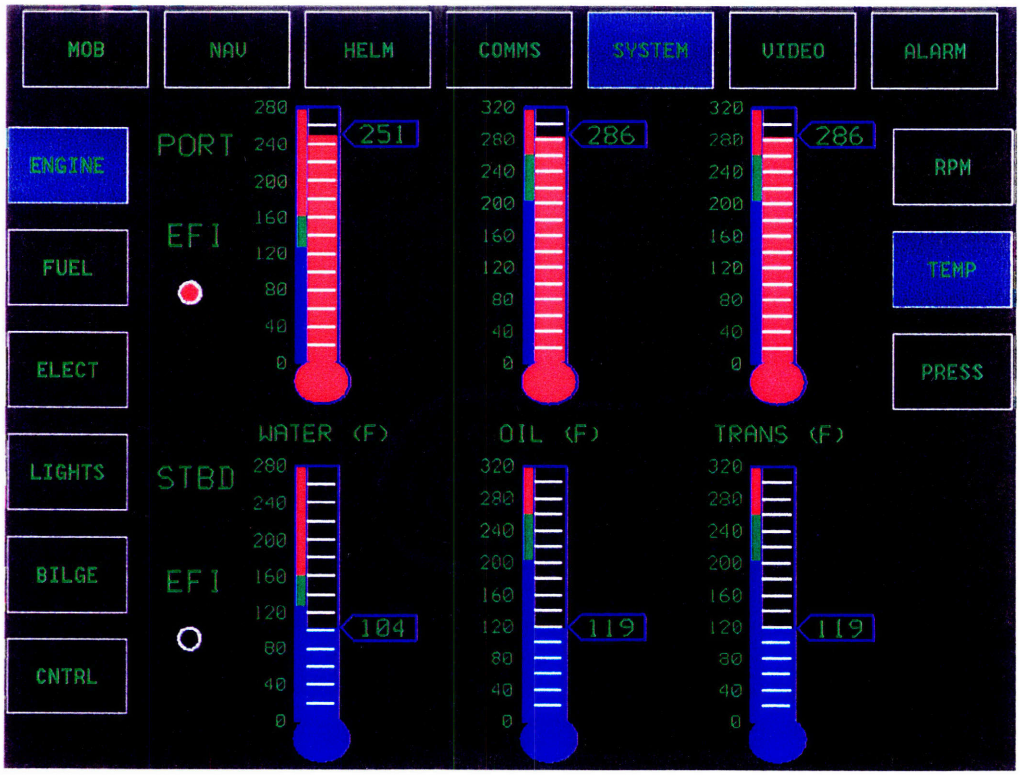

Figure 5.15 - SYSTEM Client, Engine, Temp #2

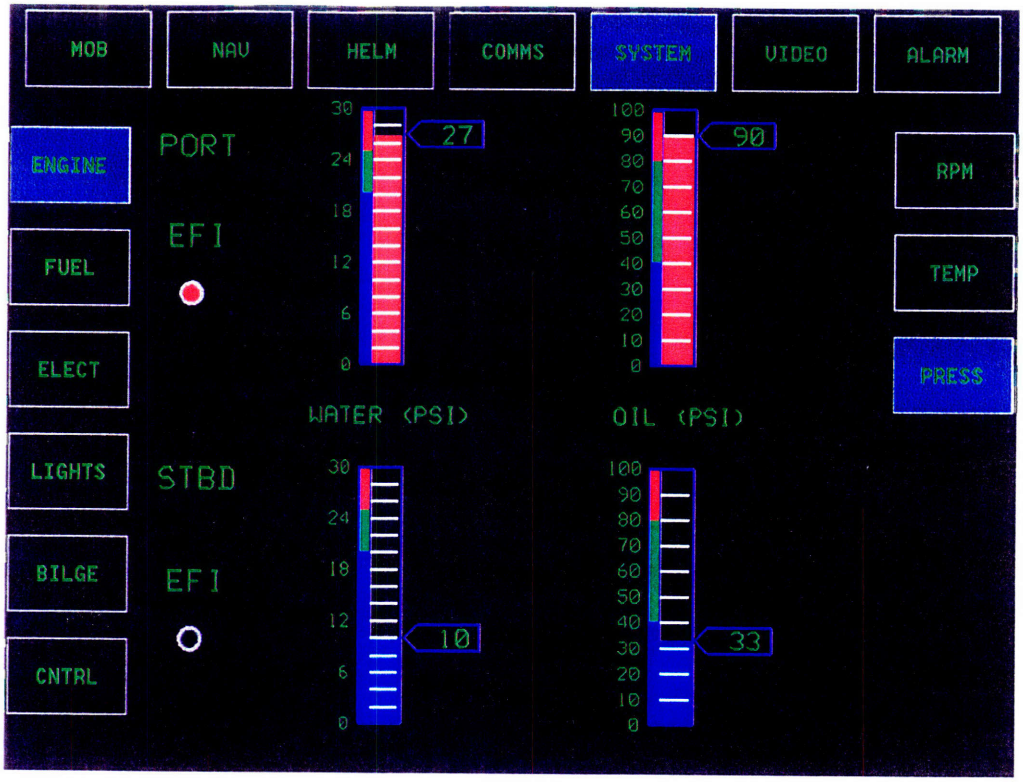

Figure 5.16 - SYSTEM Client, Engine, Pressure #1

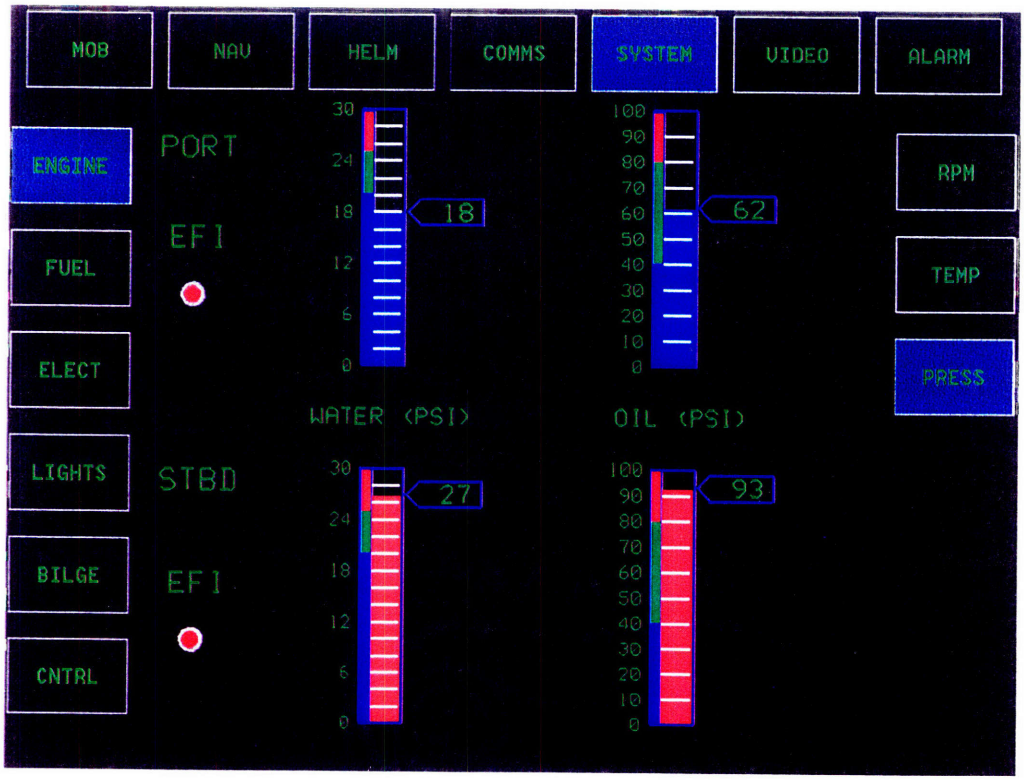

Figure 5.17 - SYSTEM Client, Engine, Pressure #2

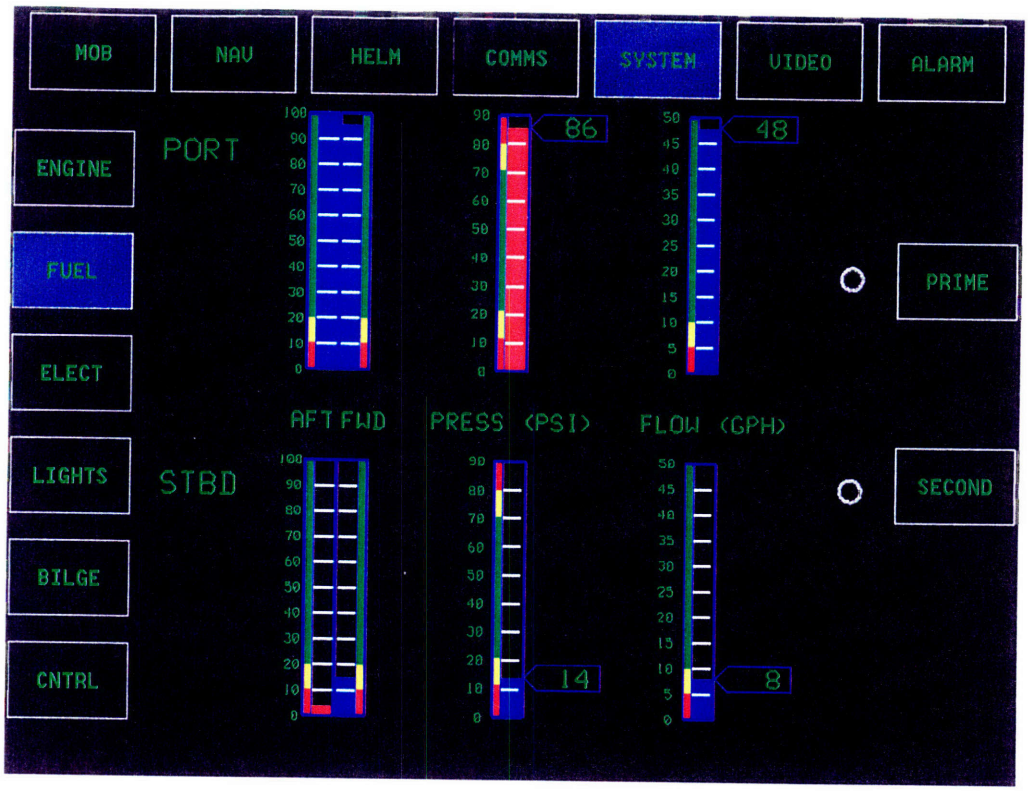

Figure 5.18 - SYSTEM Client, Fuel #1

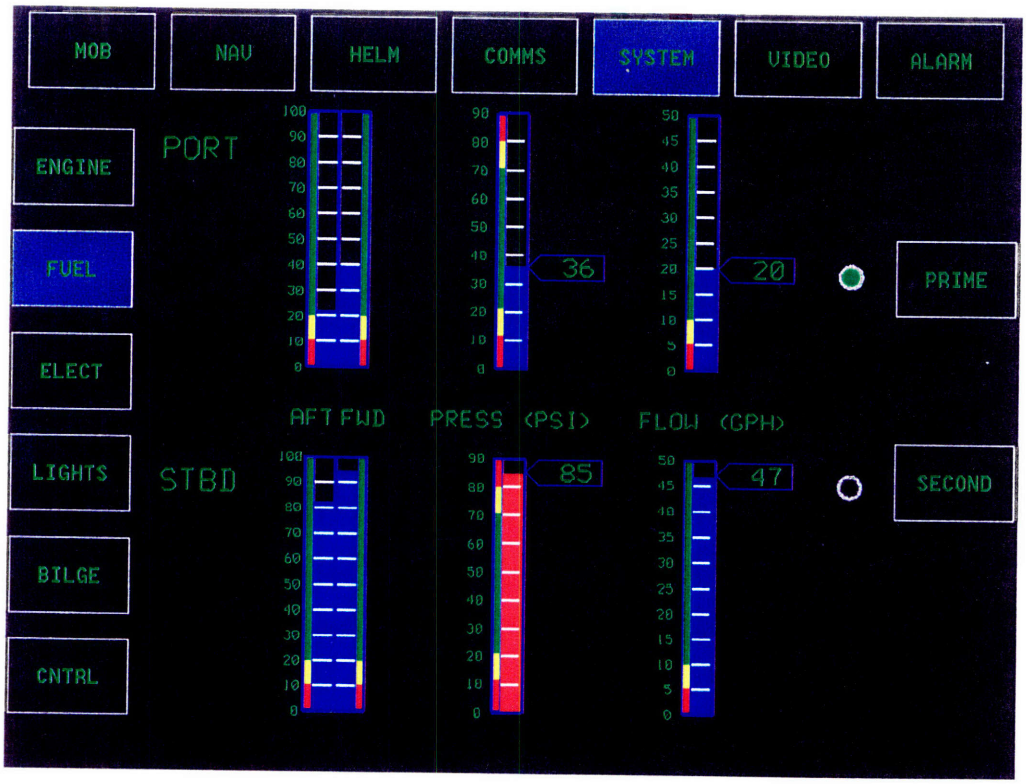

Figure 5.19 - SYSTEM Client, Fuel #2

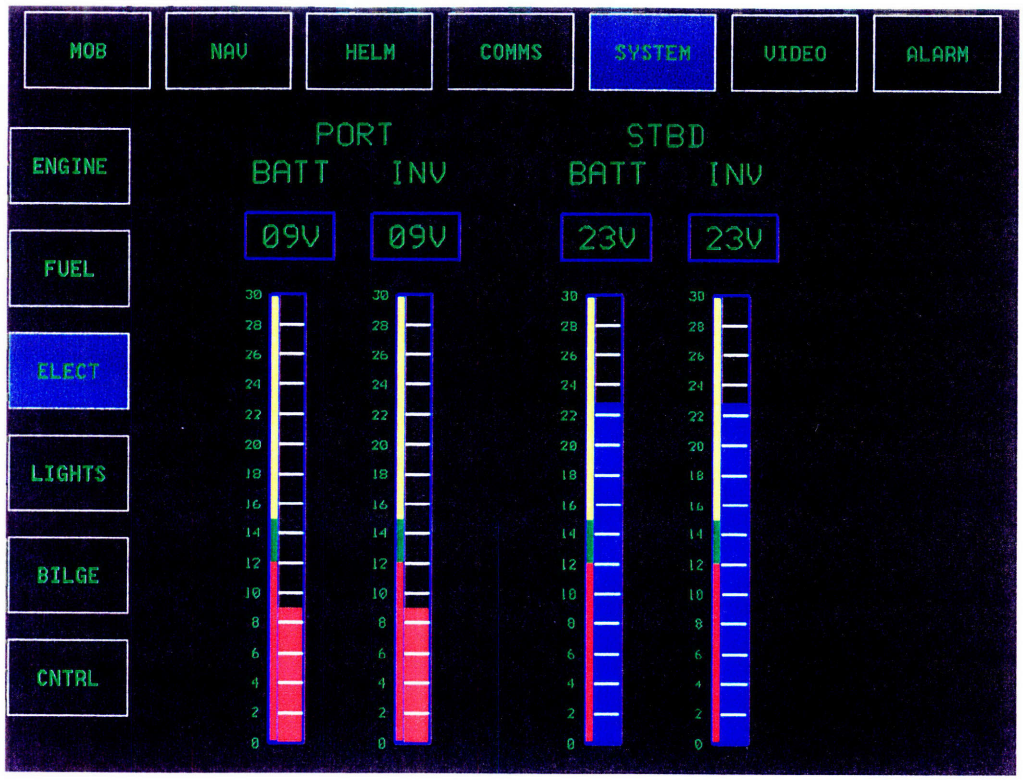

Figure 5.20 - SYSTEM Client, Electrics #1

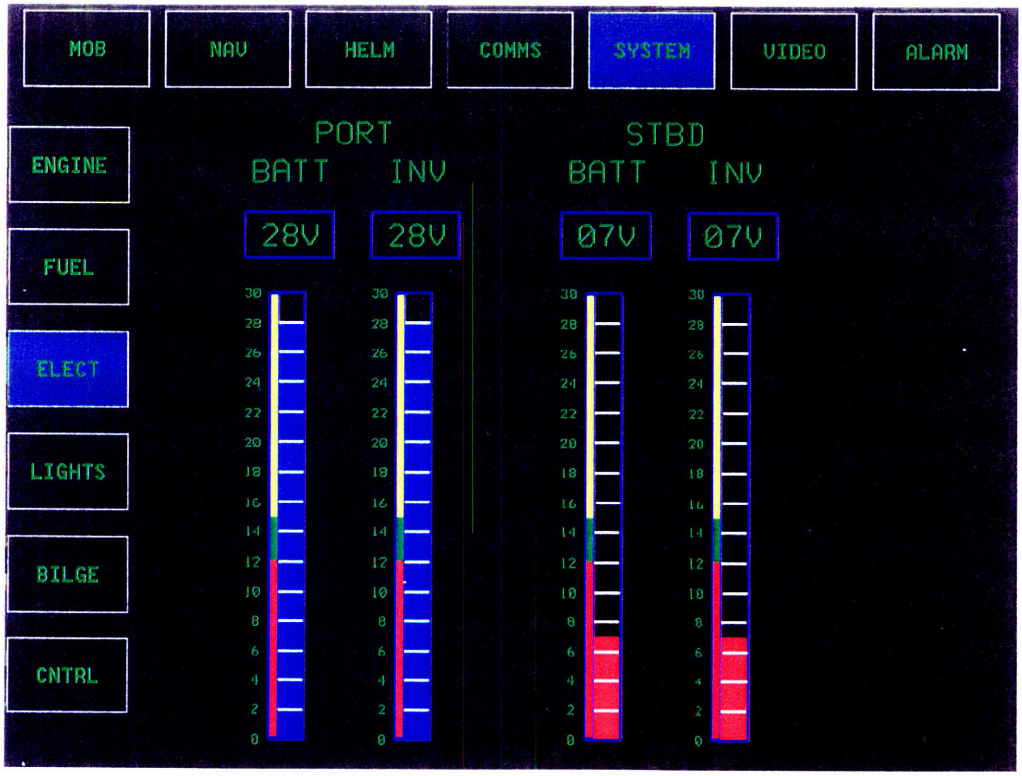

Figure 5.21 - SYSTEM Client, Electrics #2

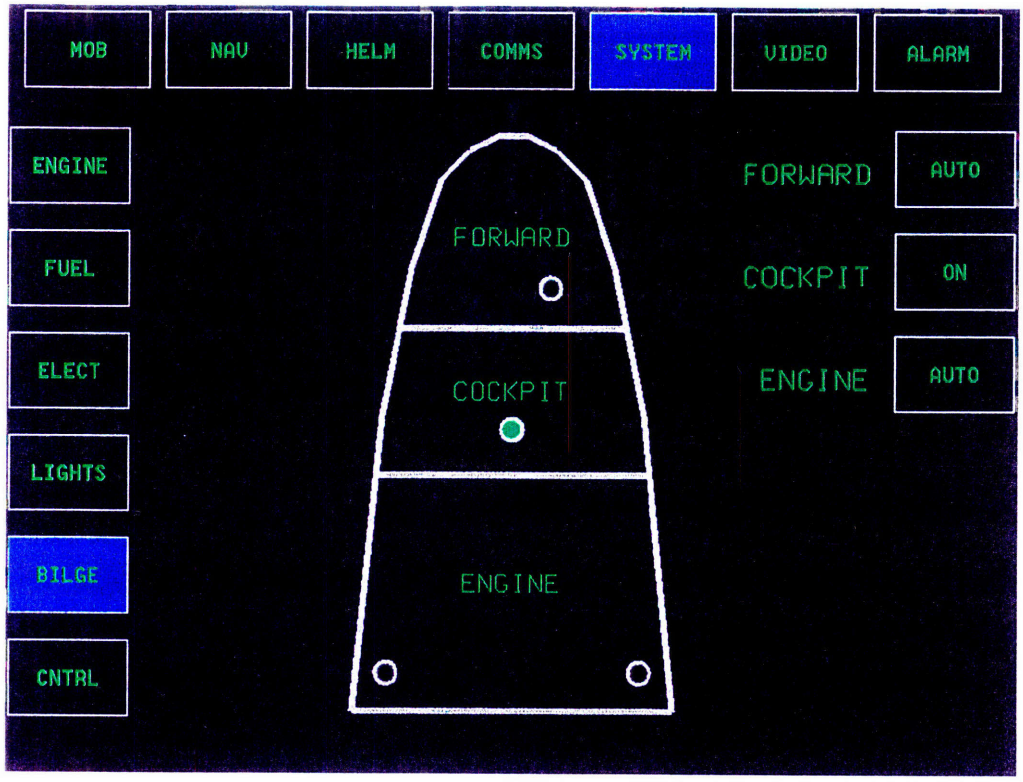

Figure 5.22 - SYSTEM Client, Bilge Pumps

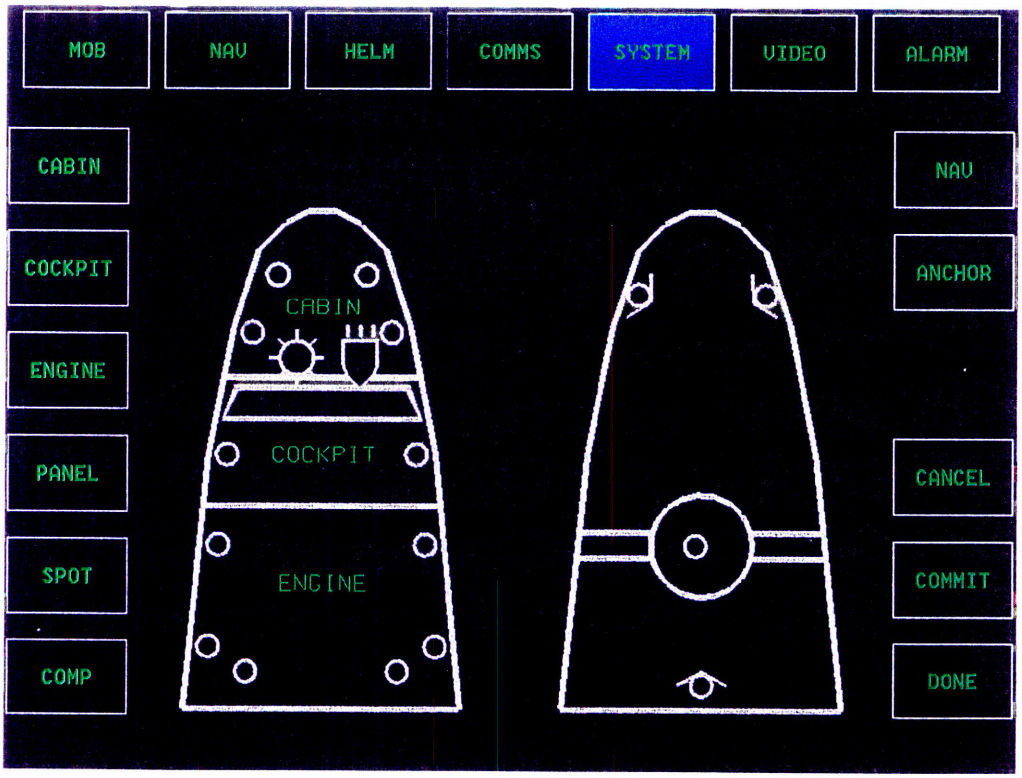

Figure 5.23 - SYSTEM Client, Lights #1

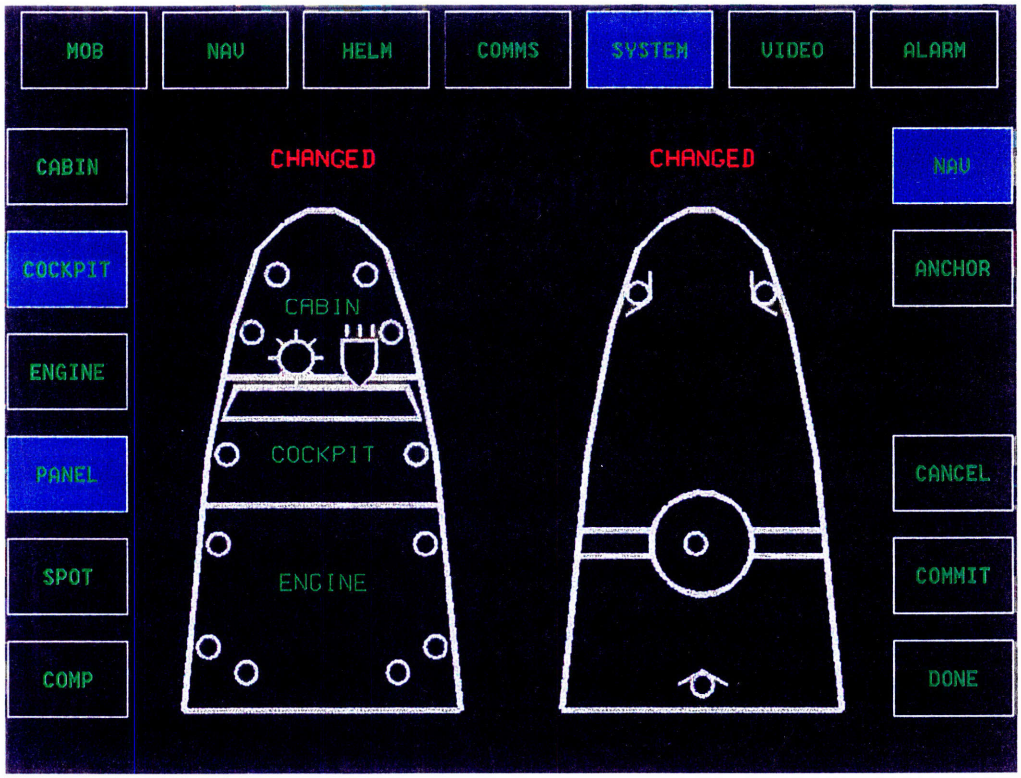

Figure 5.24 - SYSTEM Client, Lights #2

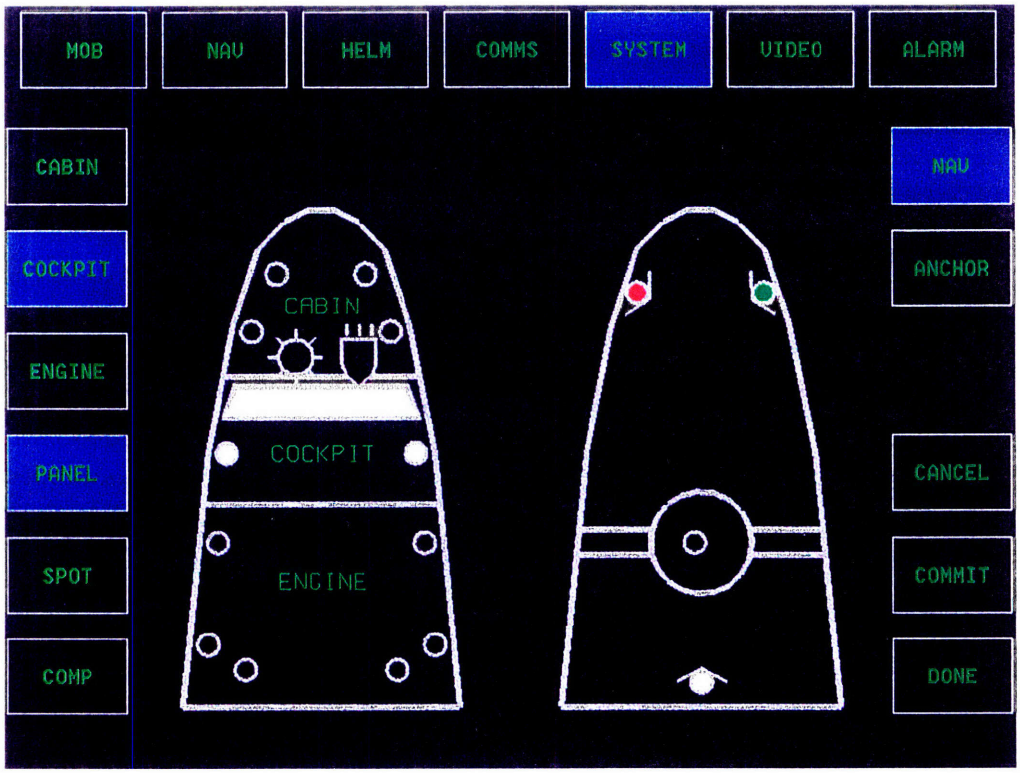

Figure 5.25 - SYSTEM Client, Lights #3

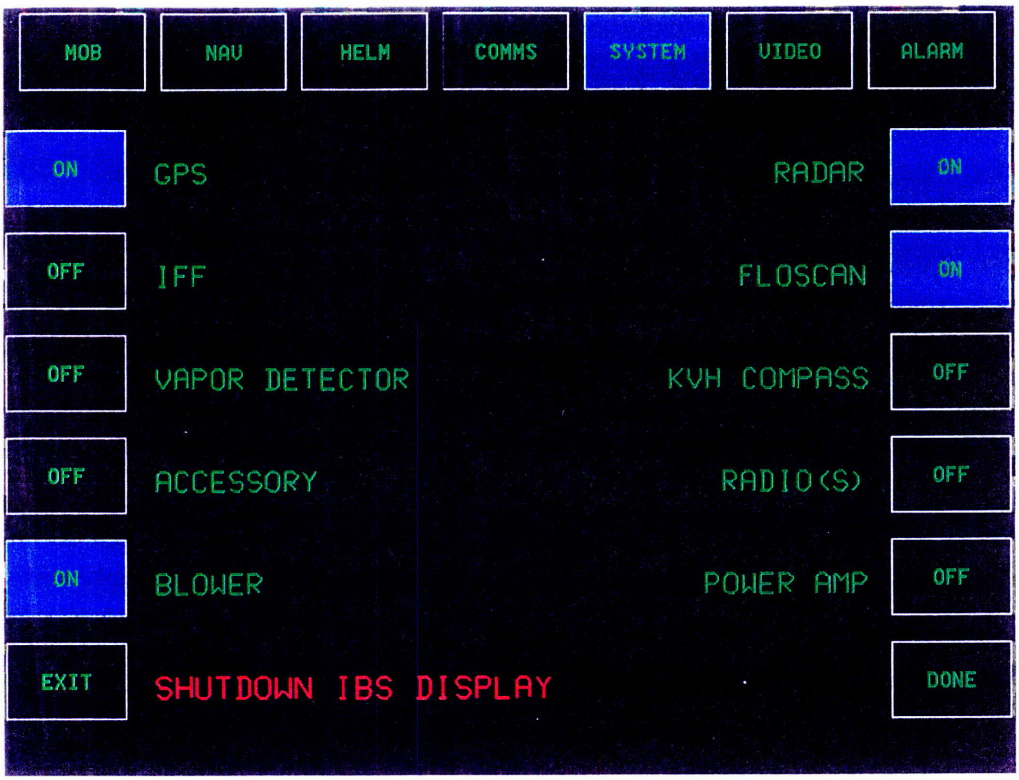

Figure 5.26 - SYSTEM Client, Controls

Figure 5.27 - SYSTEM Client, Exit Dialog

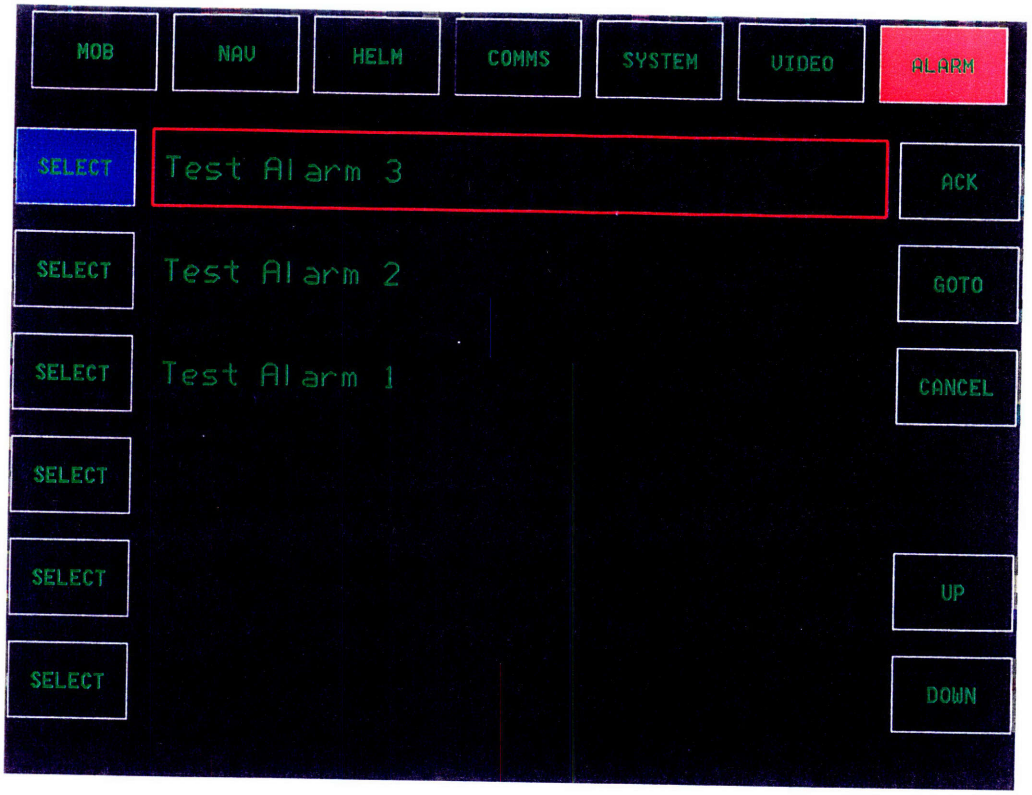

Figure 5.28 - ALARMS Client, first alarm selected

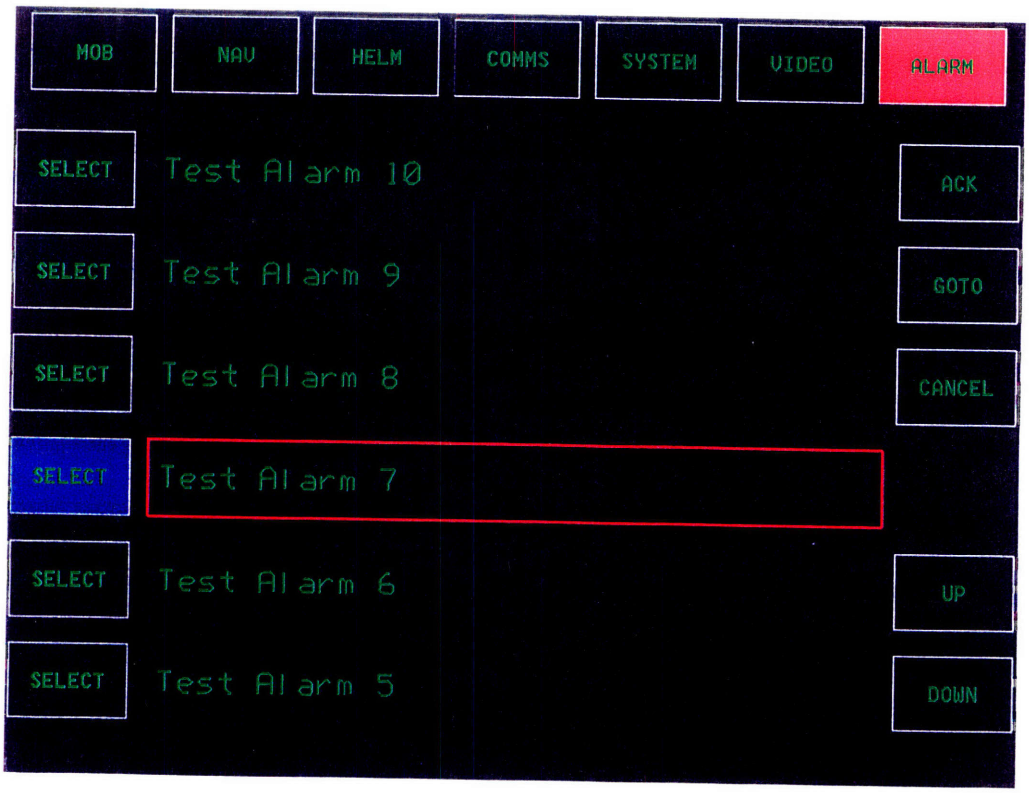

Figure 5.29 - ALARMS Client, fourth alarm selected

Chapter 6 - Evaluation and Testing

It is common in user interface design to follow a Waterfall Model beginning with a Requirements Specification phase followed by Design, Code, Test and Maintenance phases This tends to lead to an evaluation centered approach, an important step in improving the final interface Usually once the evaluation process has been reached, however, only simple changes, such as layouts, can be made Usability of the system is also an important criterion and requires a more design centered approach including interviews of users and observational studies [12] An integration of these two methods was used in the development process of IBS (See Figure 6 1)

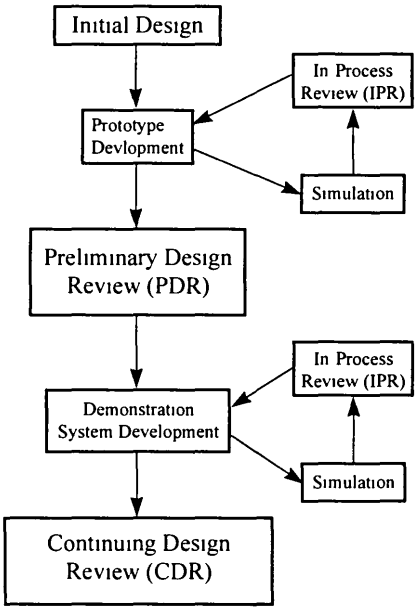

Figure 6 1 Development and Evaluation Process

Two milestones for evaluation were established for the IBS project. The first, a Preliminary Design Review (PDR) two months from the end of the Initial Design Phase, focused on a prototype GUI including a COMMS Client. Three months later, a demonstration version of the system including COMMS, HELM, and SYSTEM Clients (See Chapter 5) was delivered for a Continuing Design Review (CDR). The ALARMS Client was developed as a follow on to CDR.

In between these two larger milestones, multiple In Process Reviews (IPRs) were held to take advantage of user input. Particularly valuable methods of evaluating user interface designs [10], simulations were the primary focus of the IPRs. Two different types of simulation, one using HTML and one using Microsoft PowerPoint, were used to address different aspects of the GUI.

6.1 HTML Simulation

The first method of simulation was a series of displays in HTML format that provided a feel for the functionality of the nineteen button interface and the flow of control of the system in general. Each page contained a table, representing the buttons, with seven cells across the top and six cells in two columns on the left and right. In the middle of these cells was one large cell that held bitmaps representing the various screens of the GUI. Pages in this format were created for each screen in the system with links to subsequent

pages embedded in "button" cells. Navigation through this simulation was very similar to that of the actual system. Upon review by both staff and users, this interface was approved and work on the prototype began.

6.2 PowerPoint Simulation

Simulations of prospective screens were particularly useful in the evaluation of the IBS GUI because imagery and symbology were relatively simple. This meant that models for these screens could be developed and reviewed rapidly using a commercial drawing package. Throughout the entire development process, these simulations were reviewed by the user community and the system was revised along with their specifications.

Microsoft PowerPoint was used to create these example page layouts. With this tool, instruments such as bar graphs, radial dials, and pointers were drawn to create a snapshot image of how the final system could appear. These slides were then reviewed by both staff members and users, providing insight into user needs and perceptions tempered by designer experience.

6.3 Prototype System and Preliminary Design Review

A common method for GUI evaluation involves building a prototype of the final system [10]. Although it ignores issues of reliability or extensibility a prototype can be developed quickly and at low cost and provide insight into possible problems and misconceptions that might otherwise arise only after the system has been completed. A prototype of the GUI, including a COMMS Client was developed for PDR.

The evaluation process for PDR was a three step process. First, a demonstration of the GUI prototype interfacing with a simulated server was given to both staff and users. Following this, the GUI was integrated in a test with an engineering prototype server. The server interfaced with physical radios to provide a true end to end test. Finally, a meeting of both staff and users was convened to discuss the outcome of these demonstrations.

Several items became obvious from this phase of review. From a development point of view, it was at this point that the impact of developing the GUI Clients as separate processes and the need to manage these processes was fully realized. The Client Manager interface (See Section 5.2) was formulated to deal with this issue as well as provide for the distributed development of GUI Clients.

From a user perspective point of view, system response time was a problem that would need to be improved particularly with regard to the client/server interface.

Additionally, it was at this point that the ATM-style interface for entering radio frequencies and the design for radio presets (See Section 6.1) were created.

6.4 Development System and Continuing Design Review

The format for CDR closely resembled that for PDR using the demonstration system described in Section 4 including COMMS, HELM, and SYSTEM Clients. The GUI was connected to a simulated server to demonstrate the dynamics of the different Clients. Then, it was integrated with the demonstration server and a significant improvement in system response time over PDR was achieved. Once again creating an end to end test of the system, the server was connected to the physical pieces of hardware that will be installed on the craft, including sensors, electrical relays, and mission critical equipment. Each of these stages of the demonstration were observed by both staff and users followed by a series of meetings to discuss the results.

The GUI Clients for COMMS, HELM, and SYSTEM were approved for installation on the craft and open water testing. Specifications for the Alarm Client were outlined and development of the Client was begun, also to be tested and included in the onboard testing. Evaluation of the IBS GUI in the onboard, at-sea environment will occur during submission of this thesis and will be carried out according to [36].

Chapter 7 - Summary and Conclusions

Given the multitude of interfaces, functions, and equipment that must be continually monitored or controlled from the console, current limitations in the electro-mechanical instrumentation and control systems on the MK V SOC, HSAC, and VSV, required the development of an integrated bridge display system. IBS was conceived to address these limitations, improving crew member efficiency.

There are two main components to IBS, a central server that interfaces with the vessel's equipment and maintains all of the information about the vessel and a GUI front end, distributed across multiple clients. This architecture was chosen so that all available information could be reproduced – consistently – on all displays, minimizing the crippling effect of a damaged display.

Through the incorporation of new sensors and control pathways on the server side, IBS successfully offered all of the current and several additional bridge systems. All of these systems were tested in an end-to-end laboratory demonstration of the system.

On the client side, interaction with the system needed to be feasible in the rough physical environment on board the SEAL vessels while minimizing errors in operation and cognition. This problem was solved with an input interface consisting of nineteen bezel-mounted, configurable buttons. The ability to display changeable text freed screen space and the bezel-mounting helped to minimize mistaken inputs.

The GUI had two types of requirements. From a user's perspective, it needed to allow the crew members to focus quickly and accurately on situation specific information and to learn the system with a minimum of training. From a developer's perspective, the system needed to be general enough to be ported easily to any of the three Navy SEALs craft and to be extended with new boat systems as they are developed.

The users' requirements were satisfied through a feedback process in which many simulations and prototypes were reviewed and the results incorporated into the development process. Including user input as an integral part of this process resulted in a clear, intuitive interface for the system's displays and instrumentation.

The developers' requirements were addressed by a development framework that was easily extensible. A program manager for the individual GUI processes provided a simple interface that can be followed to develop new GUI programs, and a display toolkit provided for creation and modification of GUI screens.

Although testing on board the vessels has yet to be completed and will undoubtedly identify lingering issues with the current interface, the development process that has been followed throughout the project has helped to minimize the number and scope of these issues. The GUI framework, meanwhile, ensures that any issues that do arise can be easily and quickly addressed.

Chapter 8 - Improvements and Extensions

Although the current GUI framework offers a reasonably simple method of extending or modifying the GUI, there are a number of ways that it could be improved. Specifically, the Display Wizard could be augmented to support a large number of additional features that would simplify GUI extension.

The purpose of a GUI designer toolkit it to provide the developer with a more natural way to create a GUI. This generally means allowing the designer to work in a more visual manner [6]. In its current incarnation, the Display Wizard requires the user to type in all of the parameters for each instrument that is displayed, including attributes such as location and color. An improved toolkit would allow for direct manipulation of the objects in the display, such as drag and drop capabilities [5].

Currently, the Display Wizard is only capable of modifying visual qualities of a display. The developer must still write and compile code to handle all of the functionality that he wishes to include in the display. Instead, he could specify data variables, methods, and flows of control all within the Display Wizard using a graphical interface and a specification language and automatically generate the desired code.

Finally, if all of these features are added to the Display Wizard, online help would need to be included in the program to explain them. While external documentation is

necessary, the development environment should eliminate the need to refer to those manuals after the initial learning period [6].

One principle of user interfaces not addressed by IBS is the concept of configurability. It is generally desirable to provide the users with the ability to customize there interface [9]. In the case of IBS, user-customizable pages could be a viable solution. Take, for example, a general GUI screen that the user can optionally divide into either one, two, four, six, or eight sections (i.e. if the screen contains two slots it is divided in half, if it contains four it is quartered). Each instrument is defined to take up a certain number of slots of each size (i.e. an instrument that takes up one slot on a quartered screen would take up two on a screen cut in eighths). Each user could then choose from a set of available instruments the instruments that they would like to display on a given page. While, for the most part, this would only be possible with instruments that strictly monitor data, it would at least allow each crew member to set up personal configuration pages that include all of the instruments that they must continually monitor.

Inclusion of such a feature within IBS would necessitate a means for setting up preferences. This interface would again need to be intuitive enough for the average user to learn quickly. Additionally, a suitable set of default configurations should be created for the personal configuration pages [8].

Although it is felt that the system can be learned quickly, in order to facilitate this process, a Trainer should be developed to allow the users to train in a simulated environment with actual vessel displays. The Trainer could range anywhere from a simple desktop introduction to a full cockpit mockup with scene generation, simulated motion, and joint training maneuvers executed over a simulation network [35].

References

[1] Prepared for the Department of Defense Office of Special Technology (January 1996), *Operational Requirements Document for SOF Maritime Craft Integrated Bridge System.*

[2] Mason Woo, Jackie Neider, Tom Davis (1997): OpenGL Programming Guide, Addison-Wesley Developers Press, Reading, MA.

[3] Stephen A. Ward and Robert H. Halstead, Jr. (1990): *Computation Structures*, The MIT Press, Cambridge, MA.

[4] Charles Weicha and Stephen Boies, "Generating User Interfaces: Principles and Use of Its Style Rules," *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, Oct. 1990.

[5] Deborah Hix, "A Procedure for Evaluating Human-Computer Interface Tools," *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, Oct. 1990.

[6] Gurminder Singh and Mark Green, "Designing the Interface Designer's Interface," *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, Oct. 1990.

[7] Linda Macaulay (1995): *Human Computer Interaction for Software Designers*, International Thomson Publishing, London, UK.

[8] Wilbert O. Galitz (1994*): It's Time to Clean Your Windows: Designing GUIs that Work*, John Wiley and Sons, Inc., New York, NY.

[9] IBM (1992), *Object-Oriented Interface Design*, Que Corporation, Carmel, IN.

[10] Tony Rubin (1988), *User Interface Design for Computer Systems*, Ellis Horwood Ltd., Chichester, West Sussex, UK.

[11] Jakob Nielson (1989), *Coordinating user Interfaces for Consistency*, Academic Press, Inc.

[12] David Benyon and Philippe Palanque (1996), *Critical Issues in User Interface Systems Engineering*, Springer-Verlag London Ltd., London, UK.

[13] J. M. Reising et al, "New Cockpit Technology: Unique Opportunities for the Pilot," SPIE Vol. 2219, 1994.

[14] R. McCartney and J. Ackerman, "Primary Flight Instruments for the Boeing 777 Airplane," SPIE Vol. 2219, 1994.

[15] G. L. Neal, "Air Transport Common Cockpit," SPIE Vol. 2219, 1994.

[16] D. C. Bailey, "F-22 Cockpit Display System," SPIE Vol. 2219, 1994.

[17] G. J. Hardy, D. F. Wilkins, and R. N. Wright, "Advanced Displays for the F/A-18E/F Hornet: Application of AMLCD and Touch Sensing Technology in an Existing Tactical Fighter/Attack Crew Station," SPIE Vol. 2219, 1994.

[18] R. E. Orkis, "F-16 Retrofit Application Using Modular Avionics System Architecture and Color Active Matrix Liquid Crystal Displays," SPIE Vol. 2219, 1994.

[19] Thuan Q. Pham and Pankaj K Garg (1995): *Multithreaded Programming with Windows NT*, Prentice-Hall, Inc., Upper Saddle River, NJ.

[20] Mike Blaszczak (1996), *The Revolutionary Guide to MFC 4 Programming with Visual C++*, Wrox Press Ltd., Tyseley, Birmingham, Canada.

[21] Stanley B. Lippman (1991), *C++ Primer*, Adison-Wesley, Reading, MA.

[22] Steve Holzner (1996), *Advanced Visual C++ 4*, M&T Books, New York, NY.

[23] E. D. Gurd and C. A. Forest, "Flat Panels in Future Ground Combat Vehicles," SPIE Vol. 2734, 1996.

[24] *Paragon Mann - VSV50*, http://www.enterprise.net/unicorn/paragon/index.htm

[25] USMI (1994), *Boat Information Manual for the High Speed Assault Craft.*

[26] USMI (1995), *Boat Information Book for Mark V Special Operations Craft (SOC).*

[27] Earl L. Wiener and David C. Nagel (1989): *Human Factors in Aviation*, Academic Press, Inc., San Diego, CA.

[28] K. Wisler, J. J. Doyle Jr. and M. Giuglianotti, "Display Readability," SPIE Vol. 2219, 1994.

[29] Daniele Mariani, "Crewman's Associate Advanced Technology Demonstration," SPIE Vol. 2462, 1995.

[30] K. J. Nerius, "Comanche (RAH-66) Crew Station Display System," SPIE Vol. 2462, 1995.

[31] H. Brandtburg, "JAS 39 Cockpit Display System and Development for the Future," SPIE Vol. 2462, 1995.

[32] R. C. McKillip, "Advanced Displays and Interactive Displays Federated Laboratory," SPIE Vol. 2462, 1995.

[33] Dave Dooling, "Smoother Sailing," *IEEE Spectrum*, August 1996.

[34] Mark S. Sanders and Ernest J. McCormick (1993): *Human Factors in Engineering and Design*, McGraw-Hill, Inc., New York, NY.

[35] George Rathjens et al (1995), *Distributed Interactive Simulation of Combat,* U.S. Government Printing Office, Washington D. C.

[36] Prepared for Commander Naval Surface Warfare Center (May 1997), *Test and Evaluation Plan High Speed Assault Craft Integrated Bridge System Test Plan.*

Appendix A - GUI Client Extensibility

There are two aspects to creating new clients. As described in the previous section, the Display Wizard can be used to place all of the instruments within the Displays and configure them as desired. Afterward, here is also a small amount of code that must be written and compiled. Looking at an example in which we create a new Client with two Displays, this can be demonstrated.

First, we generate a dialog-based project in Microsoft Visual C++ [25]. For this example we will call our project "test". We then replace the generated testDlg.h and testDlg.cpp files with the ones shown below which derive testDlg from IBSClientDlg to provide our Client Shell for this Client. Note that this class has been named testDlg (by programmer preference) instead of CTestDlg, the name specified by the Microsoft Project Wizard in testDlg.h and testDlg.cpp. As a result, references to CTestDlg in other project files must be changed to testDlg. Alternately, the class defined in the files below could be changed to CTestDlg. Additionally, the default dialog for the project is generated with OK and CANCEL buttons and a static text box. These dialog items should be deleted from the dialog resource.

The files below define functions for initializing and updating the Client Shell, as well as functions for each of the left and right side buttons. Code will be added to these functions to create the test Client.

```
/////////////////////////////////////////////////////////////////////////
// testDlg.h - Class definition

#include "..\clntlib\IBSclntDlg.h"
#include "..\ibslib\ibslib.h"

/*********************************************************************/

class testDlg : public IBSclntDlg
{
// Construction
public:
      testDlg(CWnd* pParent = NULL);      // standard constructor
```

```
// Dialog Data
      //{{AFX_DATA(testDlg)
      // NOTE: the ClassWizard will add data members here
      //}}AFX_DATA

      // ClassWizard generated virtual function overrides
      //{{AFX_VIRTUAL(testDlg)
      //}}AFX_VIRTUAL

// Implementation
public:
      virtual void      DoLButton1();     // Functions for executing
      virtual void      DoLButton2();     // Client-specific left
      virtual void      DoLButton3();     // button actions
      virtual void      DoLButton4();
      virtual void      DoLButton5();
      virtual void      DoLButton6();
      virtual void      DoRButton1();     // Functions for executing
      virtual void      DoRButton2();     // Client-specific right
      virtual void      DoRButton3();     // button actions
      virtual void      DoRButton4();
      virtual void      DoRButton5();
      virtual void      DoRButton6();
      virtual void      UpdatePage();     // Function for polling server
                                          // for data.

protected:
      HICON m_hIcon;

public:
      // Generated message map functions
      //{{AFX_MSG(testDlg)
      virtual BOOL OnInitDialog();
      afx_msg HCURSOR OnQueryDragIcon();
      //}}AFX_MSG
      DECLARE_MESSAGE_MAP()
};

-------------------------------------------------------------------------------------------

/////////////////////////////////////////////////////////////////////////////
//    testDlg.cpp - Class implementation

#include "stdafx.h"
#include "test.h"
#include "testDlg.h"
#include "..\cmi_dlib\cmi_dlib.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
```

```
/*************************************************************************/

testDlg::testDlg(CWnd* pParent /*=NULL*/)
      : IBSclntDlg(pParent)
{
      //{{AFX_DATA_INIT(testDlg)
            // NOTE: the ClassWizard will add member init here
      //}}AFX_DATA_INIT
      m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}

/*************************************************************************/

BEGIN_MESSAGE_MAP(testDlg, CDialog)
      //{{AFX_MSG_MAP(testDlg)
      ON_WM_PAINT()
      ON_WM_QUERYDRAGICON()
      //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/*************************************************************************
 *** testDlg message handlers
 *************************************************************************/


BOOL testDlg::OnInitDialog()
{
      IBSclntDlg::OnInitDialog();

      // Set the icon for this dialog.  The framework does this
         automatically
      // when the application's main window is not a dialog
      SetIcon(m_hIcon, TRUE);             // Set big icon
      SetIcon(m_hIcon, FALSE);            // Set small icon
      // TODO: Add extra initialization here

      return TRUE;  // return TRUE unless you set the focus to a
                        control
}

/*************************************************************************/
// The system calls this to obtain the cursor to display while the user
drags
// the minimized window.

HCURSOR testDlg::OnQueryDragIcon()
{
      return (HCURSOR) m_hIcon;
}

/*************************************************************************/

void testDlg::DoLButton1()
{
}

/*************************************************************************/

void testDlg::DoLButton2()
```

```
{
}

/************************************************************************/

void testDlg::DoLButton3()
{
}

/************************************************************************/

void testDlg::DoLButton4()
{
}

/************************************************************************/

void testDlg::DoLButton5()
{
}

/************************************************************************/

void testDlg::DoLButton6()
{
}

/************************************************************************/

void testDlg::DoRButton1()
{
}

/************************************************************************/

void testDlg::DoRButton2()
{
}

/************************************************************************/

void testDlg::DoRButton3()
{
}

/************************************************************************/

void testDlg::DoRButton4()
{
}

/************************************************************************/

void testDlg::DoRButton5()
{
}
```

```
/*************************************************************************/

void testDlg::DoRButton6()
{
}

/*************************************************************************/

void testDlg::UpdatePage()
{
      IBSclntDlg::UpdatePage();
}
```

This is our basic Client Shell with no Displays defined. To add Displays to our Client Shell we first need to derive a testDisplay class from the base display class:

```
//////////////////////////////////////////////////////////////////////
//    testDisplay.h - Class definition

#ifndef TESTDISPLAY_H
#define TESTDISPLAY_H

#include "..\ogllib\display.h"
#include "..\ogllib\text.h"
#include "..\ogllib\barMeter.h"
#include "..\ogllib\compass.h"
#include "..\ogllib\textMeter.h"
#include "..\ogllib\multiText.h"
#include "..\ogllib\digit.h"

/*************************************************************************/

//////////////////////////////////////////////////////////////////////
// testDisplay window

class testDisplay : public display
{
// Construction
public:
      testDisplay();

// Attributes
public:

// Operations
public:

// Overrides
      // ClassWizard generated virtual function overrides
      //{{AFX_VIRTUAL(testDisplay)
      //}}AFX_VIRTUAL

// Implementation
public:
      virtual ~testDisplay();
      void UpdatePage();
```

```
      virtual void init();

      // Generated message map functions
protected:
      //{{AFX_MSG(testDisplay)
      //}}AFX_MSG
      DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////////////////////////////////////////

#endif


---------------------------------------------------------------------------------------------

/////////////////////////////////////////////////////////////////////////////
//    testDisplay.cpp - Class implementation

#include "stdafx.h"
#include "testDisplay.h"
#include "helmDlg.h"
#include "..\ogllib\instr.h"
#include "..\cmi_dlib\cmi_dlib.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/*************************************************************************/

testDisplay::testDisplay()
      :display()
{
}

/*************************************************************************/

testDisplay::~testDisplay()
{
}

/*************************************************************************/

BEGIN_MESSAGE_MAP(testDisplay, display)
      //{{AFX_MSG_MAP(testDisplay)
      ON_WM_CREATE()
      //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/*************************************************************************
 *** testDisplay message handlers
 *************************************************************************/

void testDisplay::init()
{
      display::init();
}
```

```
/*************************************************************************/
```

Now we can add two displays to our Client Shell by modifying testDlg.h. In each step in this example, the lines in bold face type are the lines that have been added to the code.

```
////////////////////////////////////////////////////////////////////////
//    testDlg.h - Class definition

#include "..\clntlib\IBSclntDlg.h"
#include "..\ibslib\ibslib.h"

#include "testDisplay.h"   // include the display definition

/*************************************************************************/

class testDlg : public IBSclntDlg
{
// Construction
public:
      testDlg(CWnd* pParent = NULL);      // standard constructor

// Dialog Data
      //{{AFX_DATA(testDlg)
      // NOTE: the ClassWizard will add data members here
      //}}AFX_DATA

      // ClassWizard generated virtual function overrides
      //{{AFX_VIRTUAL(testDlg)
      //}}AFX_VIRTUAL

// Implementation
public:
      virtual void      DoLButton1();
      virtual void      DoLButton2();
      virtual void      DoLButton3();
      virtual void      DoLButton4();
      virtual void      DoLButton5();
      virtual void      DoLButton6();
      virtual void      DoRButton1();
      virtual void      DoRButton2();
      virtual void      DoRButton3();
      virtual void      DoRButton4();
      virtual void      DoRButton5();
      virtual void      DoRButton6();
      virtual void      UpdatePage();

protected:
      HICON       m_hIcon;

      testDisplay dpy1;       // Define the
      testDisplay dpy2;       // two new Displays.
```

```
public:
      // Generated message map functions
      //{{AFX_MSG(testDlg)
      virtual BOOL OnInitDialog();
      afx_msg HCURSOR OnQueryDragIcon();
      //}}AFX_MSG
      DECLARE_MESSAGE_MAP()
};
```

Then we need to set up these Displays in the testDlg implementation in testDlg.cpp. To do this, we modify the OnInitDialog function:

```
/////////////////////////////////////////////////////////////////////////

BOOL testDlg::OnInitDialog()
{
      IBSclntDlg::OnInitDialog();

      // Set the icon for this dialog.  The framework does this
         automatically
      // when the application's main window is not a dialog
      SetIcon(m_hIcon, TRUE);              // Set big icon
      SetIcon(m_hIcon, FALSE);             // Set small icon
      // TODO: Add extra initialization here

      dpy1.antiAlias = 1;            // Optional lines to turn on
      dpy2.antiAlias = 1;            // antiAliasing for each Display

      CreateDisplay(dpy1,"test1.spec");   // These files are the data
      CreateDisplay(dpy2,"test2.spec");   // files for each Display
                                          // created with the Display
                                          // Wizard.

      return TRUE;  // return TRUE unless you set the focus to a
                       control
}

/////////////////////////////////////////////////////////////////////////
```

Now that the Displays have been created, we need to decided how the Displays will be managed by the Client Shell. For simplicity sake, we will cause the top left button to bring up dpy1 and the second left button to bring up dpy2. In order to help us do this, the IBSClntDlg class has a state variable that we can use to keep track of which Display is currently visible. Based on the state then, we can take appropriate action when the buttons are pressed.

```
/*************************************************************************/

void testDlg::DoLButton1()
{
      if (state != 1) {
            dpy2.Hide();      // If dpy1 is not the currently visible
            dpy1.Show();      // Display then make it so.
            state = 1;
      }
}
/*************************************************************************/

void testDlg::DoLButton2()
{
      if (state != 2) {
            dpy1.Hide();      // If dpy2 is not the currently visible
            dpy2.Show();      // Display then make it so.
            state = 2;
      }
}

/*************************************************************************/
```

Now, when we start this Client, we would like to buttons to reflect the fact that left button 1 is tied to dpy1 and left button 2 is tied to dpy2. We add two lines to the OnInitDialog function to take care of this:

```
/*************************************************************************/

BOOL testDlg::OnInitDialog()
{
      IBSclntDlg::OnInitDialog();

      // Set the icon for this dialog.  The framework does this
         automatically
      // when the application's main window is not a dialog
      SetIcon(m_hIcon, TRUE);             // Set big icon
      SetIcon(m_hIcon, FALSE);            // Set small icon
      // TODO: Add extra initialization here

      SetLeftFrameText(0,"DPY1","DPY2","","","","");
      SetRightFrameText(0,"","","","","","");
      // Sets the button texts.  The first argument specifies the
      // button state which we do not need to set.

      dpy1.antiAlias = 1;           // Optional lines to turn on
      dpy2.antiAlias = 1;           // antiAliasing for each Display

      CreateDisplay(dpy1,"test1.spec");   // These files are the data
      CreateDisplay(dpy2,"test2.spec");   // files for each Display
                                          // created with the Display
                                          // Wizard.

      return TRUE;  // return TRUE unless you set the focus to a
```

```
                        control
}
```

We also need to specify the initial state for the Client. In this case we will initialize the Client to show dpy1.

```
/*************************************************************************/

BOOL testDlg::OnInitDialog()
{
      IBSclntDlg::OnInitDialog();

      // Set the icon for this dialog.  The framework does this
         automatically
      // when the application's main window is not a dialog
      SetIcon(m_hIcon, TRUE);             // Set big icon
      SetIcon(m_hIcon, FALSE);            // Set small icon
      // TODO: Add extra initialization here

      SetLeftFrameText(0,"DPY1","DPY2","","","","");
      SetRightFrameText(0,"","","","","","");
      // Sets the button texts.  The first argument specifies the
      // button state which we do not need to set.

      dpy1.antiAlias = 1;           // Optional lines to turn on
      dpy2.antiAlias = 1;           // antiAliasing for each Display

      CreateDisplay(dpy1,"test1.spec");   // These files are the data
      CreateDisplay(dpy2,"test2.spec");   // files for each Display
                                          // created with the Display
                                          // Wizard.

      state = 0;                    // Set state != 1
      MenuItemHit(MENU_LEFT1);      // Emulate a left button 1 hit

      return TRUE;  // return TRUE unless you set the focus to a
                       control
}

/*************************************************************************/
```

Finally, this Client will attempt to register itself with the Client Manager when it starts. IBSClntDlg has an ID variable that specifies which top row button the Client will be registered under. We need to set this variable for this particular Client.

```
/*************************************************************************/

testDlg::testDlg(CWnd* pParent /*=NULL*/)
      : IBSclntDlg(pParent)
{
      //{{AFX_DATA_INIT(testDlg)
            // NOTE: the ClassWizard will add member init here
      //}}AFX_DATA_INIT
      m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);

      id = 3;     // Register this Client under the third
                  // top row button.
}

/*************************************************************************/
```

This is all that is necessary to launch this Client from the Client Manager and to be able to switch between its two Displays. Of course, we will want to place some instruments within these Displays. For the sake of simplicity, we will place just one Bar Meter in each Display. These Bar Meters are created with the Display Wizard and saved as the data files test1.spec and test2.spec. Although these "spec" files contain all of the information about how the Bar Meters are to be drawn, we must specify, in the code, the Data Pointers to the Data Variables that will drive them. First we will define two Data Variable:

```
//////////////////////////////////////////////////////////////////////
//    testDlg.h - Class definition

#include "..\clntlib\IBSclntDlg.h"
#include "..\ibslib\ibslib.h"

#include "testDisplay.h"   // include the display definition

/*************************************************************************/

class testDlg : public IBSclntDlg
{
// Construction
public:
      testDlg(CWnd* pParent = NULL);      // standard constructor

// Dialog Data
      //{{AFX_DATA(testDlg)
      // NOTE: the ClassWizard will add data members here
      //}}AFX_DATA

      // ClassWizard generated virtual function overrides
      //{{AFX_VIRTUAL(testDlg)
      //}}AFX_VIRTUAL
```

```
// Implementation
public:
      virtual void      DoLButton1();
      virtual void      DoLButton2();
      virtual void      DoLButton3();
      virtual void      DoLButton4();
      virtual void      DoLButton5();
      virtual void      DoLButton6();
      virtual void      DoRButton1();
      virtual void      DoRButton2();
      virtual void      DoRButton3();
      virtual void      DoRButton4();
      virtual void      DoRButton5();
      virtual void      DoRButton6();
      virtual void      UpdatePage();

protected:
      HICON       m_hIcon;

      testDisplay dpy1;       // Define the
      testDisplay dpy2;       // two new Displays.

public:

      int   bar1Val;          // Data Variables for the
      int   bar2Val;          // Bar Meters in the Displays

      // Generated message map functions
      //{{AFX_MSG(testDlg)
      virtual BOOL OnInitDialog();
      afx_msg HCURSOR OnQueryDragIcon();
      //}}AFX_MSG
      DECLARE_MESSAGE_MAP()
};
```

Now that we have the Data Variables we need to set up the Data Pointers. When we create an instrument in the Display Wizard, we are prompted for a name for that instrument. These name are then used to access the instruments within the code so that we can modify them. We will name the Bar Meters bar1 and bar2 respectively. To set up the Data Pointers for bar1 and bar2 we need to add the following lines to testDisplay.cpp:

```
/*************************************************************************/

void testDisplay::init()
{
      display::init();

      // Get a pointer to the Client Shell
      testDlg *td = (testDlg*)(GetParent());

      instr *inst;
      //GetInstr returns the instrument with the specified name
```

```
      //or NULL if none exists.
      //SetVars takes a variable length argument list allowing
      //you to set as many configuration variables as you wish.

      if (inst = GetInstr("bar1")) {
            inst->SetVars(INSTR_DATAPTR, &(td->bar1Val), INSTR_NULL);
      }
      if (inst = GetInstr("bar")) {
            inst->SetVars(INSTR_DATAPTR, &(td->bar2Val), INSTR_NULL);
      }
}

/*****************************************************************/
```

If the Display are ever going to reflect changing data, these Data Variables need to be updated over time. This is done by spawning a separate thread which calls an Update function a specified rate.

```
/*****************************************************************/

BOOL testDlg::OnInitDialog()
{
      IBSclntDlg::OnInitDialog();

      // Set the icon for this dialog.  The framework does this
         automatically
      // when the application's main window is not a dialog
      SetIcon(m_hIcon, TRUE);              // Set big icon
      SetIcon(m_hIcon, FALSE);             // Set small icon
      // TODO: Add extra initialization here

      SetLeftFrameText(0,"DPY1","DPY2","","","","");
      SetRightFrameText(0,"","","","","","");
      // Sets the button texts.  The first argument specifies the
      // button state which we do not need to set.

      dpy1.antiAlias = 1;           // Optional lines to turn on
      dpy2.antiAlias = 1;           // antiAliasing for each Display

      CreateDisplay(dpy1,"test1.spec");   // These files are the data
      CreateDisplay(dpy2,"test2.spec");   // files for each Display
                                          // created with the Display
                                          // Wizard.

      state = 0;                    // Set state != 1
      MenuItemHit(MENU_LEFT1);      // Emulate a left button 1 hit

      sleepTime = 120;              // Milliseconds between updates
      AfxBeginThread(Poller,this);  // Spawns the polling thread

      return TRUE;  // return TRUE unless you set the focus to a
                       control
}
```

Finally, the update function must be modified to retrieve the data. In general, this is accomplished through an interface library call to the Server. For this example we will assume the existence of an interface function called GetData that provides the current state for the two bars.

```
/*********************************************************************/

void testDlg::UpdatePage()
{
      GetData(&bar1Val, &bar2Val);  // Retrieves data from server

      IBSclntDlg::UpdatePage();
}

/*********************************************************************/
```

We now have a working client with two Displays that show changing Bar Meters. This is, of course, a simple example. Much more complicated functionality can be built into a Client based on the developer's needs. This example illustrates, however, just how simple it is to extend the IBS GUI.

Appendix B - IBS Interface Control Document

Version: 3.52
Last Revision: 04/08/97 by Parker

Authors:

Darryl Dietz	Draper Laboratory	617-258-1171 ddietz@draper.com
Jim Parker	Azimuth, Inc.	304-363-1162 parker@host.dmsc.net
Chris King	Draper Laboratory	617-258-1407 cmking@mit.edu

1.0 IBS Library Overview

The following document defines the IBS interface library. The IBS library provides a mechanism for multiple processes to communicate with the IBS server. To implement a flexible and easily maintainable interface, the IBS library is constructed using a Microsoft Win32 application programming interface dynamic link library (DLL) format. Using a DLL for the IBS server interface has several advantages:

- Many processes can use a single DLL.
- DLL functions can be changed without recompiling the applications that use them.
- DLL is programming language independent.

2.0 IBS Library Requirements

The IBS library is designed to meet the following requirements.

1. Provide a multiple process interface.
2. Provide data synchronization between calling processes.
3. Use C programming language function calling convention.
4. Load-time dynamic linking.
5. Compatible with Microsoft Windows NT 4.0 operating system.
6. Execute on Intel Pentium or higher processor platform.

3.0 IBS Library Files

The IBS library consists of the following files.

File	Description
ibslib.dll	IBS DLL library 1.0, executable code
ibslib.def	IBS DLL module definition file, used to create import library
ibslib.h	IBS library header file version 1.0, contains library function prototypes
ibsdtype.h	IBS data types header file, contains data structures for IBS library.

4.0 IBS Library Administration Functions

The following section defines IBS library administration functions.

This section contains the following functions:

4.1 IbsGetVersion

float IbsGetVersion(void);

IbsGetVersion function is used to determine the version number of the library.

This function returns a floating point value depicting the IBS library version number.

4.2 IbsCleanup

int IbsCleanup(void);

IbsCleanup terminates all message handling between the I/O manager and the IBS client and closes the socket connection.

This function returns an IBS_OK on success. On error, this function returns an error code. See Appendix A for possible values.

4.3 IbsStartup

int IbsStartup(void);

IbsStartup performs initialization and setup between the I/O manager and the client and establishes a socket connection for message traffic.

This function returns an IBS_OK on success. On error, this function returns an error code. See Appendix A for possible values.

5.0 Communication Functions

The following section defines the communication functions available through the IBS library.

This section contains the following functions:

5.1 IbsGetCommConfig

int IbsGetCommConfig(int comm_device, IbsCommConfig *conf);

int comm_type	- Variable containing the communication type.
IbsCommConfig *conf	- Pointer to structure containing the communication device *comm_device* configuration.

Use IbsGetCommConfig function to retrieve the current configuration for the communication device *comm_device*. This function places the current configuration for the communication device in the structure pointed to by *conf.*

The comm devices will start at zero.

The IbsCommConfig structure is defined as:

```
typedef struct {
        int comm_type;              // communication type
        int power;                  // power, 1=on 0=off
        int volume;                 // volume level, 0-99, 0=no volume
        int squelch;                // squelch level, 0-99, 0=no squelch
        int encryption;             // encryption, 1=on 0=off
        int output_pwr;             // output power level  0-100,  100=full power
        unsigned long modulation;   // modulation selected
        float rec_freq;             // receive frequency selected
        float tx_freq;              // transmit frequency selected
        int ics_monitor[5];         // radios being monitored through the ICS
        int xmit_dev;               // device operator has currently selected for transmitting
        int which;                  // what structure element was changed
} IbsCommConfig;
```

The *modulation* variable contains the modulation type selected for the communication device *comm_device*. Appendix A defines the modulation types and associated values.

The *ics_monitor* array contains the number of the radio being monitored through the ICS. A negative 1 (-1) in the array entry means there are currently not 5 radios being monitored. For example, if ics_monitor contains 2, 3, 5, -1, and -1 that would mean 3 radios are being monitored and they are numbers 2, 3, and 5.

This function returns an IBS_OK on success. On error, this function returns an error code. See Appendix A for possible values.

5.2 IbsGetCommFreqRange

int IbsGetCommFreqRange(int comm_type, unsigned long mod, IbsCommFreqRange *range);

int comm_type	- Variable containing the communication type.
unsigned long mod	- Variable containing modulation type.
IbsCommFreqRange range	- Pointer to structure containing the frequency range of communication type *comm_type* using modulation type *mod.*

Use IbsGetCommFreqRange function to retrieve the frequency range for a communication type *comm_type* configured for modulation type *mod.*

The IbsCommFreqRange structure is defined as:

```
typedef struct {
        int comm_type;              // communication type
        unsigned long modulation;   // modulation type
        float rec_freq_low;         // receive low frequency
        float rec_freq_high;        // receive high frequency
        float rec_freq_res;         // receive frequency resolution
        float tx_freq_ low;         // transmit low frequency
        float tx_freq_high;         // transmit high frequency
        float tx_freq_res;          // transmit frequency resolution
} IbsCommFreqRange;
```

The *modulation* variable contains the modulation type selected for the communication type *comm_type*. Appendix A defines the modulation types and associated values.

This function returns an IBS_OK on success. On error, this function returns an error code. See Appendix A for possible values.

5.3 IbsGetCommOptions

int IbsGetCommOptions(int comm_type, IbsCommOptions *options);

int comm_type	- Variable containing the communication type.
IbsCommOptions *options	- Pointer to structure containing the communication type *comm_device* options.

Use IbsGetCommOptions function to retrieve the options available for the communication type *comm_device*. This function places the options available to the communication type in the structure pointed to by *options*. Use these options when configuring a communication device of a certain type. See IbsSetCommConfig().

The IbsCommOptions structure is defined as:

```
typedef struct {
        int comm_type;          // communication device type
        int pwr_cont;           // power control, 1=yes 0=no
        int vol_cont;           // volume control, 1=yes 0=no
        int sqlch_cont;         // squelch control, 1=yes 0=no
        int freq_cont;          // frequency control, 1=yes 0=no
        int txrcfreq_cont;      // transmit receive frequency control, 1=yes 0=no
        int encrypt_cont;       // encryption control, 1=yes 0=no
        int outpwr_cont;        // output power control, 1=yes 0=no
        unsigned long mod_reg;  // modulation types register
} IbsCommOptions;
```

The *mod_reg* variable contains the modulation types available for the communication type *comm_type*. Appendix A defines the modulation types and associated values.

This function returns an IBS_OK on success. On error, this function returns an error code. See Appendix A for possible values.

5.4 IbsGetCommPresets

int IbsGetCommPresets(int comm_device, IbsPresets *presets);

int comm_device - Variable containing the communication device number.
IbsPresets *presets - Address pointing to the first element of an array of presets.

Use IbsGetCommPresets function to determine the preset frequencies of the communication device *comm_device*.

The comm devices will start at zero.

The IbsPresets structure is defined as:

```
typedef struct {
        float rec_freq;         // receive frequency
        float tx_freq;          // transmit frequency
} IbsPresets;
```

This function returns an IBS_OK on success. On error, this function returns an error code. See Appendix A for possible values.

5.5 IbsGetCommStatus

int IbsGetCommStatus(int comm_device, IbsCommStatus *status);

int comm_device	- Variable containing the communication device number.
IbsCommStatus *status	- Pointer to structure containing the communication type *comm_device* status.

Use IbsGetCommStatus function to determine the status of the communication type *comm_device*. This function places the status of the communication device in the structure pointed to by *status*.

The comm devices will start at zero.

The IbsCommStatus structure is defined as:

```
typedef struct {
        int comm_type;      // communication device type
        int available;      // indicates if the comm_device is available, 1=yes 0=no
        int power;          // indicates if the comm_device has power, 1=yes 0=no
} IbsCommStatus;
```

This function returns an IBS_OK on success. On error, this function returns an error code. See Appendix A for possible values.

5.6 IbsGetCommTypes

int IbsGetCommTypes(int *num_devs, int *comm_devs);

int *num_devs - Address of total number of types in the array
int *comm_devs - Address pointing to the first element of an array of communication types

IbsGetCommTypes fills the array pointed to by *comm_devs* with the communication devices currently in place on the boat. The total number of devices is defined by the value placed in *num_devs*.

For the table of communication types see Appendix A.

This function returns an IBS_OK on success. On error, this function returns an error code. See Appendix A for possible values.

NOTES:

The way this is envisioned working, each device on the boat will be able to be identified by the array index. For example, comm_devs[0] will contain the type of the first communication device on the boat, comm_devs[1] will contain the type of the second, etc. num_devs will be equal to the total number of devices installed on the boat.

5.7 IbsSetCommConfig

int IbsSetCommConfig(int comm_device, IbsCommConfig *conf);

int comm_device - Comm device to set configuration of
IbsCommConfig *conf - Structure containing the communication configuration.

Use IbsSetCommConfig function to set the current configuration for a communication device. Use the function IbsGetCommOptions to determine the options available for a communication type and the function IbsGetCommConfig to determine the current configuration of a communication device.

The comm devices will start at zero.

The IbsCommConfig structure is defined as:

```
typedef struct {
        int comm_type;              // communication type
        int power;                  // power, 1=on 0=off
        int volume;                 // volume level, 0-99, 0=no volume
        int squelch;                // squelch level, 0-99, 0=no squelch
        int encryption;             // encryption, 1=on 0=off
        int output_pwr;             // output power level 0-100, 100=full power
        unsigned long modulation;   // modulation selected
        float rec_freq;             // receive frequency selected
        float tx_freq;              // transmit frequency selected
        int ics_monitor[5];         // radios being monitored through the ICS
        int xmit_dev;               // device operator has currently selected for transmitting
        int which;                  // what structure element was changed
} IbsCommConfig;
```

The *modulation* variable contains the modulation type selected for the communication device *comm_device*. Appendix A defines the modulation types and associated values.

The *ics_monitor* array contains the number of the radio being monitored through the ICS. A negative 1 (-1) in the array entry means there are currently not 5 radios being monitored. For example, if ics_monitor contains 2, 3, 5, -1, and -1 that would mean 3 radios are being monitored and they are numbers 2, 3, and 5.

This function returns an IBS_OK on success. On error, this function returns an error code. See Appendix A for possible values.

5.8 IbsSetCommPresets

int IbsSetCommPresets(int comm_device, IbsPresets *presets);

int comm_device - Variable containing the communication device number.
IbsPresets *presets - Address pointing to the first element of an array of presets.

Use IbsSetCommPresets function to set the preset frequencies of the communication device *comm_device*.

The comm devices will start at zero.

The IbsPresets structure is defined as:

```
typedef struct {
        float rec_freq;         // receive frequency
        float tx_freq;          // transmit frequency
} IbsPresets;
```

This function returns an IBS_OK on success. On error, this function returns an error code. See Appendix A for possible values.

6.0 Helm Functions

The following section defines the helm functions available through the IBS library.

This section contains the following functions:

6.1 IbsGetAutopilotData

int IbsGetAutopilotData(IbsAutopilotData *autopilot);

IbsAutopilotData *autopilot - Pointer to structure containing the autopilot A & B information.

IbsGetAutopilotData returns the autopilot type A and type B information in the structure pointed to by *autopilot.* This is based on the APB and BWC NMEA strings.

The IbsAutopilotData structure is defined as:

```
typedef struct {
        char utc[10];                   // UTC of observation  (19:23:32)

        // From APB string...

        char xte_mag[8];                // Magnitude of cross track error
        char steer_dir[2];              // Direction to steer (L or R)
        char status_arrival[2];         // A = arrival circle entered
        char steer_heading[8];          // Heading to steer to destination waypoint
        char steer_mag_or_true[2];      // M = magnetic,  T = true

        // From BWC string...

        char waypoint_id[20];           // Destination waypoint ID
        char waypoint_lat[9];           // Waypoint latitude (ddmm.mm)
        char waypoint_lat_dir[2];       // Waypoint lat direction (N/S)
        char waypoint_long[10];         // Waypoint longitude (dddmm.mm)
        char waypoint_long_dir[2];      // Waypoint long direction (E/W)
        char bearing_true[10];          // Bearing true to waypoint
        char bearing_mag[10];                   // Bearing magnetic to waypoint
        char distance_nm[10];           // Distance in nautical miles
} IbsAutopilotData;
```

The BWC string returns time (UTC), distance, bearing, and location of a specified waypoint from present position. Therefore, the bearing_true in this structure is the bearing to waypoint_id which is located at the coordinates waypoint_lat and waypoint_long. The distance_nm is distance in nautical miles to the same waypoint.

This function returns an IBS_OK on success. On error, this function returns an error code. See Appendix A for possible values.

6.2 IbsGetCurrentWPInfo

int IbsGetCurrentWPInfo(IbsCurrentWPInfo *waypointinfo);

IbsCurrentWPInfo *waypointinfo - Pointer to structure containing the waypoint information.

IbsGetCurrentWPInfo returns the current waypoint information in the structure pointed to by *waypointinfo*. This is based on the BOD and XTE NMEA strings.

The IbsCurrentWPInfo structure is defined as:

```
typedef struct {

        // From BOD string...

        char bod_bearing_true[10];          // Bearing true
        char bod_bearing_mag[10];           // Bearing magnetic
        char bod_dest_waypoint_id[20];      // Destination waypoint ID
        char bod_orig_waypoint_id[20];      // Origin waypoint ID

        // From XTE string...

        char status1[2];                    // A = data valid, V = Loran-C blink
        char status2[2];                    // A = data valid, V = Loran-C cycle lock warning
        char error_mag[10];                 // Magnetic cross track error
        char steer_dir[2];                  // Direction to steer (L/R)
} IbsCurrentWPInfo;
```

This function returns an IBS_OK on success. On error, this function returns an error code. See Appendix A for possible values.

6.3 IbsGetHelm

int IbsGetHelm(IbsHelmInfo *helminfo);

IbsHelmInfo *helminfo - Pointer to structure containing the boat helm information.

IbsGetHelm returns the boat's helm information in the structure pointed to by *helminfo*. Due to the fact that this information is obtained through the parsing of NMEA ASCII strings and no calculations will be performed on the data, it has been left in character string format. This is based on the GLL, VTG, VHW, and WPL NMEA strings.

GLL, VTG, and VHW all provide information about the boat. WPL provides information about the latitude and longitude of waypoint_id.

The IbsHelmInfo structure is defined as:

```
typedef struct {

    // GLL string...

    char latitude[9];           // Latitude     (ddmm.mm)
    char lat_dir[2];                    // Latitude direction (N/S)
    char longitude[10];         // Longitude   (dddmm.mm)
    char long_dir[2];           // Longitude direction (E/W)

    // VTG string...

    char course_ground_true[8]; // Course over ground degrees true
    char course_ground_mag[8]; // Course over ground degrees magnetic
    char speed_ground_knots[8]; // Speed relative to ground in knots
    char speed_ground_kmhr[8]; // Speed relative to ground in km/hr

    // VHW string...

    char heading_true[8];       // Heading degrees true from GPS
    char heading_mag[8];        // Heading degrees magnetic from GPS

    // Actual magnetic compass reading...

    char compass_heading[6];    // Heading from compass (0 - 359)

    // WPL string...

    char waypoint_lat[9];       // Waypoint latitude  (ddmm.mm)
    char waypoint_lat_dir[2];   // Waypoint latitude direction (N/S)
```

```
    char waypoint_long[10];      // Waypoint longitude  (dddmm.mm)
    char waypoint_long_dir[2];   // Waypoint longitude direction (E/W)
    char waypoint_id[20];        // Waypoint identifier

    char depth[6];               // Depth of water in feet
} IbsHelmInfo;
```

This function returns an IBS_OK on success. On error, this function returns an error code. See Appendix A for possible values.

6.4 IbsGetTabSettings

int IbsGetTabSettings(float *port_drivetab, float *port_trimtab, float *star_drivetab,
float *star_trimtab);

float *port_drivetab - Port drive tab setting
float *port_trimtab - Port trim tab setting
float *star_drivetab - Starboard drive tab setting
float *star_trimtab - Starboard trim tab setting

IbsGetTabSettings returns the current positions for the drivetab and the trimtab. The values returned are in the range of 0 to 9.

This function returns an IBS_OK on success. On error, this function returns an error code. See Appendix A for possible values.

6.5 IbsGetWaypointIDs

int IbsGetWaypointIDs(IbsWaypointIDs *waypoints);

IbsGetWaypointIDs *waypoints - Pointer to structure containing the waypoint identifiers.

IbsGetWaypointIDs returns the waypoint identifiers for route number zero in the structure pointed to by *waypoints*. This is based on the R00 NMEA string. NOTE: This function only returns the 14 identifiers for route number zero.

The IbsWaypointIDs structure is defined as:

```
typedef struct {
        int num_ids;                    // number of waypoint identifiers (0-14)
        char id[14][20];                // array of waypoint identifiers
} IbsWaypointIDs;
```

This function returns an IBS_OK on success. On error, this function returns an error code. See Appendix A for possible values.

7.0 Engine Functions

The following section defines the engine functions available through the IBS library.

This section contains the following functions:

7.1 IbsGetEngineAlarmRanges

int IbsGetEngineAlarmRanges(int engine_num, IbsEngineAlarmRanges *alarmRanges);

int engine_num - Which engine to check. Use the defines:
ibsD_PORT_ENGINE 0
ibsD_STAR_ENGINE 1

IbsEngineAlarmRanges *alarmRanges - Pointer to structure containing the engine alarm range information.

IbsGetEngineAlarmRanges returns the engine alarm range data in the structure pointed to by *alarmRanges.*

The IbsEngineAlarmRanges structure is defined as:

```
typedef struct {
        int oil_press_low;          // Oil pressure low value
        int oil_press_hi;           // Oil pressure high value
        int oil_temp_low;           // Oil temperature low value
        int oil_temp_hi;            // Oil temperature high value
        int water_press_low;        // Water pressure low value
        int water_press_hi;         // Water pressure high value
        int water_temp_low;         // Water temperature low value
        int water_temp_hi;          // Water temperature high value
        int trans_temp_low;         // Transmission temperature low value
        int trans_temp_hi;          // Transmission temperature high value
        int batt_volt_low;          // Battery voltage low value
        int batt_volt_hi;           // Battery voltage high value
        int inverter_volt_low;      // Inverter voltage low value
        int inverter_volt_hi;       // Inverter voltage high value
        int fuel_press_low;         // Fuel pressure low value
        int fuel_press_hi;          // Fuel pressure high value
        int fuel_flow_low;          // Fuel flow low value
        int fuel_flow_hi;           // Fuel flow high value
        int for_tank_level_low;     // Forward tank fuel level low value
        int for_tank_level_hi;      // Forward tank fuel level high value
        int aft_tank_level_low;     // Aft tank fuel level low value
        int aft_tank_level_hi;      // Aft tank fuel level high value
        int tach_low;               // Tachometer low value
        int tach_hi;                // Tachometer high value
} IbsEngineAlarmRanges;
```

This function returns an IBS_OK on success. On error, this function returns an error code. See Appendix A for possible values.

7.2 IbsGetEngineData

int IbsGetEngineData(int engine_num, IbsEngineData *enginedata);

int engine_num - Which engine to check. Use the defines:
ibsD_PORT_ENGINE 0
ibsD_STAR_ENGINE 1
IbsEngineData *enginedata - Pointer to structure containing the engine data information.

IbsGetEngineData returns the engine data in the structure pointed to by *enginedata.*

The IbsEngineData structure is defined as:

```
typedef struct {
        int oil_press;              // Oil pressure  (0 - 100 psi)
        int oil_temp;               // Oil temperature  (0 - 320 F)
        int water_press;            // Water pressure  (0 - 30 psi)
        int water_temp;             // Water temperature  (0 - 280 F)
        int trans_temp;                     // Transmission temperature  (0 - 320 F)
        int batt_volt;              // Battery voltage  (0 - 30 V)
        int inverter_volt;          // Inverter voltage (0 - 30 V)
        int EFI_diagnostic;         // EFI diagnostic (1 or 0)
        int fuel_press;             // Fuel pressure  (0 - 90 psi)
        int fuel_flow;              // Fuel flow  (0 - 50 gph)
        int for_tank_level;         // Forward tank fuel level  (0 - 90 gal)
        int aft_tank_level;         // Aft tank fuel level  (0 - 90 gal)
        int tach;                   // Tachometer (0 - 8000 rpm)
} IbsEngineData;
```

This function returns an IBS_OK on success. On error, this function returns an error code. See Appendix A for possible values.

7.3 IbsGetEngineDataRanges

int IbsGetEngineDataRanges(int engine_num, IbsEngineDataRanges *dataRanges);

int engine_num - Which engine to check. Use the defines:
ibsD_PORT_ENGINE 0
ibsD_STAR_ENGINE 1

IbsEngineDataRanges *dataRanges - Pointer to structure containing the engine data sensor range information.

IbsGetEngineDataRanges returns the engine data sensor ranges in the structure pointed to by *dataRanges.*

The IbsEngineDataRanges structure is defined as:

```
typedef struct {
        int oil_press_low;          // Oil pressure low value
        int oil_press_hi;           // Oil pressure high value
        int oil_temp_low;           // Oil temperature low value
        int oil_temp_hi;            // Oil temperature high value
        int water_press_low;        // Water pressure low value
        int water_press_hi;         // Water pressure high value
        int water_temp_low;         // Water temperature low value
        int water_temp_hi;          // Water temperature high value
        int trans_temp_low;         // Transmission temperature low value
        int trans_temp_hi;          // Transmission temperature high value
        int batt_volt_low;          // Battery voltage low value
        int batt_volt_hi;           // Battery voltage high value
        int inverter_volt_low;      // Inverter voltage low value
        int inverter_volt_hi;       // Inverter voltage high value
        int fuel_press_low;         // Fuel pressure low value
        int fuel_press_hi;          // Fuel pressure high value
        int fuel_flow_low;          // Fuel flow low value
        int fuel_flow_hi;           // Fuel flow high value
        int for_tank_level_low;     // Forward tank fuel level low value
        int for_tank_level_hi;      // Forward tank fuel level high value
        int aft_tank_level_low;     // Aft tank fuel level low value
        int aft_tank_level_hi;      // Aft tank fuel level high value
        int tach_low;               // Tachometer low value
        int tach_hi;                // Tachometer high value
} IbsEngineDataRanges;
```

This function returns an IBS_OK on success. On error, this function returns an error code. See Appendix A for possible values.

8.0 Light Functions

The following section defines the light functions available through the IBS library.

This section contains the following functions:

8.1 IbsGetLightOptions

int IbsGetLightOptions(unsigned long *opt_reg);

unsigned long *opt_reg - Value associated with controlling particular lights.

IbsGetLightOptions returns 0 if nothing can be controlled by IBS, 9 if lights 4 and 1 can be controlled, etc. See Appendix A for light values table.

This function returns an IBS_OK on success. On error, this function returns an error code. See Appendix A for possible values.

8.2 IbsGetLightStatus

int IbsGetLightStatus(unsigned long *opt_reg);

unsigned long *opt_reg - Status of particular lights.

IbsGetLightStatus returns a 0 if no lights are on, a 9 if lights 4 and 1 are on, etc. See Appendix A for light values table.

This function returns an IBS_OK on success. On error, this function returns an error code. See Appendix A for possible values.

8.3 IbsSetLightStatus

int IbsSetLightStatus(unsigned long toggle_reg);

unsigned long toggle_reg - Toggles the state of specified lights.

IbsSetLightStatus toggles the state of the light(s) indicated by toggle_reg. If toggle_reg is 0 then do not change the state of any light, if toggle_reg is 9 then toggle the state of light 4 and light 1 -- if light is on, turn it off; if light is off, turn it on. See Appendix A for light values table.

This function returns an IBS_OK on success. On error, this function returns an error code. See Appendix A for possible values.

9.0 Bilge Functions

The following section defines the bilge functions available through the IBS library.

This section contains the following functions:

9.1 IbsGetBilgeOptions

int IbsGetBilgeOptions(unsigned long *opt_reg);

unsigned long *opt_reg - Value associated with controlling bilge pumps.

IbsGetBilgeOptions returns 0 if nothing can be controlled by IBS, 5 if pumps 1, 3, and 4 can be controlled, etc. See Appendix A for bilge pump values table.

This function returns an IBS_OK on success. On error, this function returns an error code. See Appendix A for possible values.

9.2 IbsGetBilgeStatus

int IbsGetBilgeStatus(unsigned long *status_reg);

unsigned long *status_reg - Status of particular pumps.

IbsGetBilgeStatus returns a 0 if no pumps are on, a 5 if pumps 1, 3, and 4 are on, etc. See Appendix A for bilge pump values table.

This function returns an IBS_OK on success. On error, this function returns an error code. See Appendix A for possible values.

9.3 IbsSetBilgeStatus

int IbsSetBilgeStatus(unsigned long active_reg);

unsigned long active_reg - Activates specified pumps.

IbsSetBilgeStatus activates the pump(s) indicated by active_reg. If active_reg is 0 then do not activate any pump, if active_reg is 5 then activate pumps 1, 3, and 4. See Appendix A for bilge pump values table.

This function returns an IBS_OK on success. On error, this function returns an error code. See Appendix A for possible values.

10.0 Relay Functions

The following section defines the relay functions available through the IBS library.

This section contains the following functions:

10.1 IbsGetRelayOptions

int IbsGetRelayOptions(unsigned long *relay_reg);

unsigned long *relay_reg - Value associated with controlling relay switches.

IbsGetRelayOptions returns 0 if nothing can be controlled by IBS, 9 if relays 4 and 1 can be controlled, etc. See Appendix A for relay values table.

This function returns an IBS_OK on success. On error, this function returns an error code. See Appendix A for possible values.

10.2 IbsGetRelayStatus

int IbsGetRelayStatus(unsigned long *relay_reg);

unsigned long *relay_reg - Status of particular relays.

IbsGetRelayStatus returns a 0 if no relays are on, a 9 if relays 4 and 1 are on, etc. See Appendix A for relay values table.

This function returns an IBS_OK on success. On error, this function returns an error code. See Appendix A for possible values.

10.3 IbsSetRelayStatus

int IbsSetRelayStatus(unsigned long active_reg);

unsigned long active_reg - Activates specified relays.

IbsSetRelayStatus activates the relay(s) indicated by active_reg. If active_reg is 0 then do not activate any relay, if active_reg is 9 then activate relays 4 and 1. See Appendix A for relay values table.

This function returns an IBS_OK on success. On error, this function returns an error code. See Appendix A for possible values.

11.0 Alarm Functions

The following section defines the alarm functions available through the IBS library.

This section contains the following functions:

11.1 IbsAckAlarm

int IbsAckAlarm(int id);

int id - Alarm id.

IbsAckAlarms acknowledges alarm *id*. It removes alarm from list and calls AlarmUpdate(0) to tell client to refresh the alarm list. This sets the alarm trigger level n units higher or lower depending on whether an upper bound or lower bound was violated.

See Appendix A for a list of possible alarm ID values.

This function returns an IBS_OK on success. On error, this function returns an error code. See Appendix A for possible values.

11.2 IbsCancelAlarm

int IbsCancelAlarm(int id);

int id - Alarm id.

IbsCancelAlarm removes the alarm *id* from list and calls AlarmUpdate(0) to tell client to refresh the alarm list.

See Appendix A for a list of possible alarm ID values.

This function returns an IBS_OK on success. On error, this function returns an error code. See Appendix A for possible values.

11.3 IbsGetAlarms

int IbsGetAlarms(IbsAlarm *alarmList);

unsigned long active_reg - Activates specified relays.

IbsGetAlarms sets *alarmList* to the head of the list of currently active alarms.

The IbsAlarm structure is defined as:

```
typedef struct  alarmStruct{
        int id;                         // Alarm ID
        char msg[109];                  // Message - 3 lines, 35 characters per line
        int gotoAction;                         // 3 digits specifying button hits for top, left, and
right button                                    // sets respectively.  On "goto" this sequence of
buttons will
                                        // be pressed.  A zero indicates no button press.  (i.e. 300
                                        // indicates only the third top row button will be pressed).
        int type;                       // Future use (alarm priorities perhaps?)
        struct alarmStruct *next;       // Pointer to next item in linked list
} IbsAlarm;
```

See Appendix A for a list of possible alarm ID values.

This function returns an IBS_OK on success. On error, this function returns an error code. See Appendix A for possible values.

11.4 AlarmUpdate

void AlarmUpdate(int new);

int new - Indicates a new alarm has occurred.

AlarmUpdate signals the client, telling it to refresh its alarm list by calling IbsGetAlarms(). If a new alarm is being signalled, the *new* parameter is set to 1.

Appendix A

TABLES

COMM TYPES*

Comm Type	Value	Description
IBSCOM_ICS	0x01	
IBSCOM_PTPA	0x02	
IBSCOM_MST-ICOM	0x03	
IBSCOM_PRC-138	0x04	
IBSCOM_LST-5C	0x05	

ERROR CODES*

Error	Value	Description
IBS_OK	0x01	Everything is fine
IBS_SERVER_NO_REPLY	0x02	No answer from server
IBS_COMMDEV_NO_REPLY	0x03	No answer from comm device

MOD TYPES*

Type	Value	Description
IBSMOD_AM	0x0001	AM voice modulation
IBSMOD_FM	0x0002	FM voice modulation
IBSMOD_FM_FSK	0x0004	FM-FSK modulation - 16000 bps
IBSMOD_AM_ASK	0x0008	AM-ASK modulation - 16000 bps
IBSMOD_BPSK	0x0010	BPSK shaped modulation - 1200 and 2400 bps
IBSMOD_DBPSK	0x0020	DBPSK modulation - shaped differentially encoded 1200 and 2400 bps
IBSMOD_BEACON	0x0040	Beacon/Guard modulation - tone 1500 to 300 Hz
IBSMOD_J3E	0x0080	J3E modulation - single sideband, upper or lower, suppressed carrier telephony
IBSMOD_H3E	0x0100	H3E modulation - compatible AM single sideband plus full carrier
IBSMOD_J2A	0x0200	J2A modulation - CW single sideband suppressed carrier
IBSMOD_F3E	0x0400	F3E modulation - FM telephony

WHICH VALUES

Type	Value	Description
ibsD_RXFREQ	0x0001	
ibsD_TXFREQ	0x0002	
ibsD_MODE	0x0004	
ibsD_SQUELCH	0x0008	
ibsD_VOLUME	0x0010	
ibsD_ENCRYP	0x0020	
ibsD_OUTPWR	0x0040	
ibsD_POWER	0x0080	
ibsD_MONITOR	0x0100	
ibsD_RADVOL	0x0200	

LIGHT VALUES

Type	Value	Description
ibsD_LIGHT01	0x0001	Indicates light 1
ibsD_LIGHT02	0x0002	Indicates light 2
ibsD_LIGHT03	0x0004	Indicates light 3
ibsD_LIGHT04	0x0008	Indicates light 4
ibsD_LIGHT05	0x0010	Indicates light 5
ibsD_LIGHT06	0x0020	Indicates light 6
ibsD_LIGHT07	0x0040	Indicates light 7
ibsD_LIGHT08	0x0080	Indicates light 8

BILGE PUMP VALUES

ibsD_BILGE1	0x0001	Bilge pump 1
ibsD_BILGE2	0x0002	Bilge pump 2
ibsD_BILGE34	0x0004	Bilge pumps 3 & 4

RELAY VALUES

ibsD_POWER_AMP_RELAY	0x0001	RF Power amplifier
ibsD_RADIOS_RELAY	0x0002	Radios
ibsD_RADAR_RELAY	0x0004	RADAR
ibsD_GPS_RELAY	0x0008	GPS
ibsD_COMPASS_RELAY	0x0010	Compass
ibsD_FLOSCAN_RELAY	0x0020	Floscan
ibsD_PRIMEFUEL_RELAY	0x0040	Primary fuel pump
ibsD_SECONDFUEL_RELAY	0x0080	Secondary fuel pump
ibsD_VAPOR_RELAY	0x0100	Vapor detector
ibsD_ACCESS_RELAY	0x0200	Accessory
ibsD_BEACON_RELAY	0x0400	IFF Beacon
ibsD_BLOWER_RELAY	0x0800	Blower

ALARM ID VALUES

ibsD_GENERIC_ALARM	0x0000	Catch-all alarm message	?,?,?
ibsD_OIL_PRESS_HI	0x0001	Oil pressure too high	5,1,3
ibsD_OIL_PRESS_LOW	0x0002	Oil pressure too low	5,1,3
ibsD_OIL_TEMP_HI	0x0003	Oil temperature too high	5,1,2
ibsD_OIL_TEMP_LOW	0x0004	Oil temperature too low	5,1,2
ibsD_WATER_PRESS_HI	0x0005	Water pressure too high	5,1,3
ibsD_WATER_PRESS_LOW	0x0006	Water pressure too low	5,1,3
ibsD_WATER_TEMP_HI	0x0007	Water temperature too high	5,1,2
ibsD_WATER_TEMP_LOW	0x0008	Water temperature too low	5,1,2
ibsD_TRANS_TEMP_HI	0x0009	Transmission temperature too high	5,3,0
ibsD_TRANS_TEMP_LOW	0x000A	Transmission temperature too low	5,3,0
ibsD_BATT_VOLT_HI	0x000B	Battery voltage too high	5,3,0
ibsD_BATT_VOLT_LOW	0x000C	Battery voltage too low	5,3,0
ibsD_INV_VOLT_HI	0x000D	Inverter voltage too high	5,2,0
ibsD_INV_VOLT_LOW	0x000E	Inverter voltage too low	5,2,0
ibsD_FUEL_PRESS_HI	0x000F	Fuel pressure too high	5,2,0
ibsD_FUEL_PRESS_LOW	0x0010	Fuel pressure too low	5,2,0
ibsD_FUEL_FLOW_HI	0x0011	Fuel flow too high	5,2,0
ibsD_FUEL_FLOW__LOW	0x0012	Fuel flow too low	5,2,0
ibsD_FOR_TANK_HI	0x0013	Forward fuel tank too high	5,2,0
ibsD_FOR_TANK_LOW	0x0014	Forward fuel tank too low	5,2,0
ibsD_AFT_TANK_HI	0x0015	Aft fuel tank too high	5,2,0
ibsD_AFT_TANK_LOW	0x0016	Aft fuel tank too low	5,2,0
ibsD_TACH_HI	0x0017	Tachometer too high	5,1,1
ibsD_TACH_LOW	0x0018	Tachometer too low	5,1,1
ibsD_MAN_OVERBOARD	0x0019	Man overboard	2,0,0
ibsD_LIGHT1_CB	0x001A	Light 1 circuit breaker	5,4,0
ibsD_LIGHT2_CB	0x001B	Light 2 circuit breaker	5,4,0
ibsD_LIGHT3_CB	0x001C	Light 3 circuit breaker	5,4,0
ibsD_LIGHT4_CB	0x001D	Light 4 circuit breaker	5,4,0
ibsD_LIGHT5_CB	0x001E	Light 5 circuit breaker	5,4,0
ibsD_LIGHT6_CB	0x001F	Light 6 circuit breaker	5,4,0
ibsD_LIGHT7_CB	0x0020	Light 7 circuit breaker	5,4,0
ibsD_LIGHT8_CB	0x0021	Light 8 circuit breaker	5,4,0
ibsD_BILGE1_CB	0x0022	Bilge 1 circuit breaker	5,5,0
ibsD_BILGE2_CB	0x0023	Bilge 2 circuit breaker	5,5,0
ibsD_BILGE34_CB	0x0024	Bilges 3 & 4 circuit breaker	5,5,0
ibsD_BLOWER_CB	0x0025	Blower circuit breaker	5,6,0
ibsD_IFFBEACON_CB	0x0026	IFF beacon circuit breaker	5,6,0
ibsD_ACCESS_CB	0x0027	Accessory circuit breaker	5,6,0

ibsD_VAPOR_CB	0x0028	Vapor detector circuit breaker	5,6,0
ibsD_SECOND_FUEL_CB	0x0029	Secondary fuel pump circuit breaker	5,2,0
ibsD_PRIME_FUEL_CB	0x002A	Primary fuel pump circuit breaker	5,2,0
ibsD_FLOSCAN_CB	0x002B	Floscan circuit breaker	5,6,0
ibsD_COMPASS_CB	0x002C	Compass circuit breaker	5,6,0
ibsD_GPS_CB	0x002D	GPS circuit breaker	5,6,0
ibsD_RADAR_CB	0x002E	RADAR circuit breaker	5,6,0
ibsD_RADIOS_CB	0x002F	Radios circuit breaker	5,6,0
ibsD_POWER_AMP_CB	0x0030	RF power amplifier circuit breaker	5,6,0

*Table entries incomplete